Jesse Schell　陈笑雨　庄宏
作序推荐



Unity实战（第2版）

使用C#开发多平台游戏

[美] 约瑟夫•霍金 (Joseph Hocking)　著
蔡俊鸿　译

MANNING

清华大学出版社

Unity 实战

(第 2 版)

[美] 约瑟夫·霍金(Joseph Hocking) 著
蔡俊鸿 译

清华大学出版社
北 京

Joseph Hocking
Unity in Action，Second Edition
EISBN: 978-1-61729-496-9

北京市版权局著作权合同登记号　图字：01-2018-3782

图书在版编目(CIP)数据

Unity 实战：第 2 版 / (美)约瑟夫 • 霍金(Joseph Hocking) 著；蔡俊鸿 译. —北京：清华大学出版社，2019

书名原文：Unity in Action, Second Edition

ISBN 978-7-302-51895-2

Ⅰ. ①U…　Ⅱ. ①约…　②蔡…　Ⅲ. ①游戏程序－程序设计　Ⅳ. ①TP311.5

中国版本图书馆 CIP 数据核字(2018)第 294422 号

责任编辑：王　军　于　平
封面设计：孔祥峰
版式设计：思创景点
责任校对：牛艳敏
责任印制：宋　林

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市国英印务有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　印　　张：22　　字　　数：444 千字
版　　次：2019 年 1 月第 1 版　　印　　次：2019 年 1 月第 1 次印刷
定　　价：69.80 元

产品编号：077874-01

推 荐 序 一

关于艺术，但凡有点“私心”的人都想跟它沾上边。有人把它作为茶余饭后的谈资，有人作为业余消遣的娱乐，有人视其为谋生的工具……有人浅尝辄止，有人不求甚解，有人半途折返，有人情之所至……无论你是基于哪一种出发点，都希望自己能够仰高山之巅、充分实现自我。

进入移动游戏时代后，游戏作为第九艺术，不仅给众多玩家带来了电影级别的剧情体验以及美妙的音乐享受，亦带给玩家们现实生活中难以实操的情感体验与交互。游戏作为文化产品所带来的视听享受以及超越电影的交互优势，使得玩家群体以难以置信的速度在扩大，这是游戏最好的时代，但也是游戏开发者面临巨大挑战的时代。

随着玩家群体审美提升，游戏的复杂度、可玩性、渲染效果以及性能面临着巨大的挑战。Unity3D 作为用于移动游戏开发的现代化引擎，凭借其强大的跨平台开发能力，在当前的移动游戏时代中应运而生，具有强大的竞争力。凭借其出色的渲染能力，在游戏中为视觉效果贡献巨大力量，此外，由于其友好的可视化开发环境，又使之成为移动游戏开发的不二选择。

本书的上一版《Unity5 实战　使用 C#和 Unity 开发多平台游戏》已成为行业内认可度颇高的优秀专业书籍。身边很多技术开发者，在进入移动游戏开发前都通过这本书学习 Unity3D 开发的相关知识，他们均赞不绝口。我与译者在多个游戏项目中都有合作，译者本人就是一位非常资深的技术专家，也是使用 Unity3D 进行移动游戏开发的先驱。他对于游戏开发始终抱有巨大的热情，对于开发技术始终精益求精，对于艺术由始至终情之所至。我相信译者这样的技术态度，足以让《Unity 实战(第 2 版)》的译文更加精确无误且更易于理解与推广。现在，您手上阅读的这本书已是第 2 版，我相信本书不仅仅是第 1 版精彩的延续，更是在此基础上的超越。相信译者本人，即是相信自己，希望你在本书中能拾你所想，得你所思，如你所愿。

陈笑雨

——360 游戏艺术副总裁

推荐序二

作为一本以"实战"为名的技术类专业书籍，本书的第 1 版《Unity5 实战 使用 C#和 Unity 开发多平台游戏》此前在亚马逊上连续数月居于同类书籍销量之冠，结合身边诸多从业者阅读后的赞誉推荐，已经证明了自身的价值。这里的价值既反映出了原书作者的丰富经验和工程智慧，也体现了译者极强的专业素养。值此再版之际，应译者邀请，为此作序，深感荣幸。

在行业中深植多年，深知追求艺术性或专业性乃至两者合一的难能可贵。因为我们时常会在概念上把工程、设计和管理进行融合，但在实践上又把它们分离，并且将具体的研发方式归结于各种流派，将诸多的不可控因素归结于风格、习惯的区别或者环境使然。以各种限定的模式去适配知识的获取途径与种类，从而降低了知识应有的价值。本书很好地解决了我们作为游戏开发者为何而读书的问题。因为这种实战派的传授方式，相当于一次言传身教，使得我们很容易理解技术。

游戏作为特殊类型的软件，为逻辑性和设计性的结合体，兼具理性和感性的部分，在面向用户层体现得更加清晰。Unity3D 作为主流的商用引擎，功能强大，易用性好，并且具有较完善的工具链和较丰富的扩展，也让本书所传递的知识具有更广泛的受众群体。而随着引擎的升级换代，并不会影响本书的价值，这种经验的传承性，作者和译者在文字中所传递的钻研和分析，结合引擎发展所面临的问题以及解决思路，都能为读者在遇到本书范畴以外的问题时获得启发，此谓举一反三。

与译者相识数载，知其作为一名资深的行业技术专家，有过多年多产的研发履历，从未离开过游戏开发的一线。本书虽为他业余时间的译作，亦是一番呕心沥血，未敢一丝一毫怠慢。因此，诸君读此书，应有所得，或得他山之石，或得其中金玉。

庄　宏

——GameArk 副总裁

译　者　序

Unity 是当今最炙手可热的游戏开发工具之一，它是轻松创建诸如三维视频游戏、建筑可视化、实时三维动画的综合型游戏开发平台，是一个全面整合的专业游戏引擎。它可发布运行在 Windows、Mac、iPhone、Windows Phone 8 和 Android 平台上的游戏，也可以利用插件发布网页游戏。很多著名的游戏，如神庙逃亡、新仙剑、QQ 乐团等，都出自这个平台。

Unity 有以下几项优点：第一，Unity 的部署很简单，还自带一个 IDE：MonoDevelop，只要按下 install，之后的创建新项目、多平台打包等操作均可以在编辑器中直接完成。第二，Unity 形成了一个规模化的插件市场，在此基础上，Unity 具有相当多的中间件，可以大大加快独立开发者和公司的开发进度。第三，Unity 的社区是当前各种游戏开发社区中最活跃的，这点可以从“知乎”上的 Unity 3D 话题的关注人数看出。第四，C#作为脚本可以在编程效率和运行效率之间取得比较好的平衡，使用 C#，今后的微软一系列新技术也很有可能会和 Unity 搭配。

Unity 有大量有价值的学习资源，但这些资源比较零散，需要进行深度挖掘才能找到需要的内容。而本书把初学者需要了解的所有内容都放在一个地方，以清晰、富有逻辑的方式呈现出来，为初学者打开了游戏编程的大门。尤其是本书的“实践”部分，读者很快就可以开始编写代码——不只是编写书中的示例代码，还编写自己的游戏代码，因为本书并不仅仅简单完成游戏示例所需的功能代码，还经常对代码进行重构，提升可扩展性和复用性。

本书分为三部分，第 I 部分介绍跨平台的游戏开发环境 Unity，演示在 3D 中编写移动示例的步骤，再将移动示例转变为第一人称射击游戏，讲解射线发射和基础 AI，最后导入和创建美术资源。第 II 部分学习如何在 Unity 中创建 2D 益智游戏，接着用平台游戏机制扩展 2D 游戏，介绍 Unity 中最新的 GUI 功能，展示如何在 3D 中创建第三人称移动游戏，阐述如何在游戏中实现交互设备和物品。第III部分讨论如何与互联网通信，如何编写音频功能，如何将不同章节的碎片整合到一个游戏中，最后构建最终应用，发布到多个平台。本书最后还提供了 4 个附录，分别介绍了场景导航、外部工具、Blender 和学习资源。

本书针对的读者群是对 Unity 很陌生的编程老手，以及游戏开发新手。

掌握了本书的内容后，可以关注《Unity 圣典》和《Unity 用户手册》，把在初学阶段忽略的内容进行选择性的补充学习。再进一步，可以关注 Unity 社区、Unity Answers、Unity Wiki 和“知乎”的 Unity 板块，此时，要对 Unity 的各种细节问题、优化、底层原理和新的技术方案进行深入思考和系统学习。

在此要感谢清华大学出版社的编辑，他们为本书的翻译投入了巨大的热情并付出了很多心血。没有他们的帮助和鼓励，本书不可能顺利付梓。

对于这本经典之作，译者本着“诚惶诚恐”的态度，在翻译过程中力求“信、达、雅”，但是鉴于译者水平有限，错误和失误在所难免，如有任何意见和建议，请不吝指正。本书主要章节由蔡俊鸿翻译，参与翻译的还有陈妍、何美英、陈宏波、熊晓磊、管兆昶、潘洪荣、曹汉鸣、高娟妮、王燕、谢李君、李珍珍、王璐、王华健、柳松洋、曹晓松、陈彬、洪妍、刘芸、邱培强、高维杰、张素英、颜灵佳、方峻、顾永湘、孔祥亮。

译　者

第一版的赞誉

“本书简明扼要，示例突出。作为一名新用户，我发现本书是一个无价之宝。”

—Dan Kacenjar Sr.，基石软件

“所有的障碍都消失了，我很快就把游戏从概念变成构建好的软件。”

—Philip Taffet, SOHOsoft LLC

“很快就能让游戏运转起来。”

—Sergio Arbeo, codecantor

“涵盖了有效使用 Unity 的所有关键元素。”

—Shiloh Morris，南内华达州水务局

“推荐给所有使用 Unity 开始游戏编程的人。”

—Alex Lucas，独立承包商

“使用干净的代码教学并说明了如何修改代码，以获得更有趣的结果。”

—亚马逊读者

第一版序

我在 1982 年就开始进行游戏编程。那时候很困难，因为没有互联网。资源只有少数糟糕的书籍和杂志，里面的代码片段虽然吸引人，却很混乱，而且根本就没有游戏引擎！编写游戏代码是一场艰巨的战斗。

非常羡慕今天的读者可以阅读《Unity 5 实战 使用 C#和 Unity 开发多平台游戏》，Unity 引擎为许多人打开了游戏编程的大门。Unity 达到了一个很好的平衡：一方面，Unity 已经成为一个强大、专业的游戏引擎；另一方面，Unity 仍然是初学者负担得起的、易于接近的游戏引擎。

“易于接近”指的是通过恰当的引导可以很快上手。有一次，我参加了一个由魔术师运营的马戏团。魔术师对我非常友善，帮助我成为一名出色的表演者。魔术师说：“当你站在舞台上时，需要许下承诺：‘我不会浪费你们的时间’”。

我最喜欢本书的 “实践”部分。作者没有浪费读者的宝贵时间，读者很快就开始编写代码——不是编写无意义的代码，而是可以理解和构建的有趣代码，因为他知道，读者不只是想阅读本书，不只是想编写书中的示例——还想编写自己的游戏。

在本书的指导下，读者上手的速度远超自己的期望。请随着 Joseph 的步伐学习，在准备好之后，不要羞于抛弃他的学习路线，去规划自己的学习路线。跳到最感兴趣的部分——尝试实验，请大胆而勇敢地进行尝试！如果你迷失了方向，还可以返回到《Unity 实战(第 2 版)》中。

不必在此序中浪费时间——游戏在等着你开发！在日历上标记一下今天的日期，因为从今天开始要发生翻天覆地的变化。永远记住今天是你开始制作游戏的第一天。

Jesse Schell
Schell Games 的 CEO
The Art of Game Design 一书的作者

前　　言

我从事游戏编写工作很长时间了，但最近才开始使用 Unity。当我开始开发游戏时，Unity 尚未出现，它的第 1 版在 2005 年发布。从一开始，它就承诺要作为游戏开发工具，但直到发布了几个版本，它也没有实现诺言。iOS 和 Android(统称为“移动”平台)等平台是后来才出现的，这些平台在很大程度上促成了 Unity 日益突出的地位。

最初，我将 Unity 视为一个有趣的开发工具，我关注它，但并不真正使用它。那段时间，我在为桌面计算机、网站编写游戏，为各种客户端开发项目。我使用过 Blitz3D 和 Flash 等工具，它们很适合编程，但有诸多限制。随着这些工具开始衰落，我就一直在寻找开发游戏的更好方式。

我从 Unity 3 开始体验，后来在 Synapse Games 进行开发时就完全转向了 Unity。最初是为 Synapse 开发网页游戏，最终全面转向移动游戏。那时我们进行游戏开发的整个生命周期，因为 Unity 允许从同一个代码库部署到网页和移动平台！

我一直认为分享知识很重要，讲授游戏开发课程也有好几年了。这么做的主要原因是很多导师和老师的表率作用。我在多所学校授课，一直想写一本关于游戏开发的书籍。

本书的许多方面都是我第一次学习 Unity 时所期望包含的教学内容。Unity 的众多优点之一是有大量有价值的学习资源，但这些资源比较零散(诸如脚本参考或独立的教程)，需要读者进行深度挖掘才能找到需要的内容。最好有一本书，把需要了解的所有内容都放在一个地方，以清晰、合乎逻辑的方式呈现出来，这就是本书的目标。本书针对的读者群是对 Unity 很陌生的编程老手，以及游戏开发新手。项目的选择反映了我快速连续地完成各种自由职业项目而获得技能和信心的经验。

学习使用 Unity 开发游戏是一场激动人心的冒险。对我来说，学习如何开发游戏意味着要忍受很多麻烦。但对读者而言，拥有了本书则意味着拥有了一份清晰简明的学习资源。

致　　谢

我要感谢 Manning 出版社给了我撰写本书的机会。与我共事的编辑，包括 Robin de Jongh、Dan Maharry，帮助我完成了这个任务，本书也因为他们的反馈更加出色。Candace West 担任第 2 版的主编。我真诚地感谢在开发和出版本书时与我一起共事的人。

我的写作受益于每一位审稿人的审查。感谢 Alex Lucas、Craig Hoffman、Dan Kacenjar、Joshua Frederick、Luca Campobasso、Mark Elston、Philip Taffet、Rene van den Berg、Sergio Arbeo Rodriguez、Shiloh Morris、Victor M. Perez、Christopher Haupt、Claudio Caseiro、David Torribia Inigo、Dean Tsaltas、Eric Williams、Nickie Buckner、Robin Dewson、Sergey Evsikov 和 Tanya Wilke。

特别感谢技术开发编辑 Scott Chaussee 和技术校对员 Christopher Haupt、Rene van den Berg 和 Shiloh Morris 对本书第 2 版进行了审核。我还要感谢 Jesse Schell 为我的书作序。

接下来，我要感谢给予我丰富 Unity 经验的相关人员。首先要感谢的是 Unity Technologies(制作 Unity 游戏引擎的公司)。我很感激 gamedev.stackexchange.com 社区。我几乎每天都会访问那个 QA 站点，向其他人学习并回答问题。促使我使用 Unity 的最大动力来自 Alex Reeve，我在 Synapse Games 的老板。同样，我从同事那里学到了一些技巧和技术，它们都展现在我编写的代码中。

最后，我要感谢我的妻子 Virginia，感谢她在我写这本书时给予我的支持。直到我开始写这本书，才真正明白这个项目在我的生活中占据了多大的位置，对周围的人产生了多大的影响。非常感谢她的爱和鼓励。

关 于 本 书

本书介绍如何使用 Unity 编写游戏。有经验的程序员可以把它当成 Unity 的入门书籍。本书的目标十分明确：带领有一些编程经验但没有 Unity 经验的读者使用 Unity 开发游戏。

讲授开发最好的方式是完成示例项目，学生通过制作示例来学习，这正是本书采用的方式。本书的各个主题展现为构建游戏示例的步骤，当浏览本书时，鼓励读者在 Unity 中构建这些游戏。每几章挑选不同的项目来讲解，而不是整本书只开发一个项目。其他有些书籍采用“一个完整项目”的方法讲解，不足之处是如果对前面的章节不感兴趣，就很难跳到中间的章节。

本书比大多数 Unity 书籍(特别是入门书籍)有更严格的编程内容。如果不知道如何编写计算机程序，最好先使用 Codecademy 之类的资源学习，在学会如何编写程序之后再回到本书。

不要担心具体的编程语言，本书大量使用了 C#，也可以使用其他语言的技能。本书的第 I 部分会花时间介绍新的概念，会小心谨慎、一步一步地在 Unity 中开发第一款游戏，但剩下的章节将更快速地推进，让读者了解多个游戏类型。本书最后会描述部署到各种平台(如 Web 和移动平台)，但本书的主旨不会提及最终的部署目标，因为 Unity 与平台无关。

至于游戏开发的其他方面，广泛覆盖的美术学科会稀释本书涵盖的 Unity 知识，加大 Unity 外部软件(例如，所使用的动画软件)的比重。关于美术任务的讨论将仅限于 Unity 或所有游戏开发者都应该知道的方面。

学习路线图

第 1 章　介绍跨平台的游戏开发环境——Unity。学习 Unity 中任何对象所基于的组件系统原理，介绍如何编写和运行基本脚本。

第 2 章　演示在 3D 中编写移动示例的步骤，涵盖鼠标和键盘输入等主题。全面解释 3D 位置和旋转的定义和管理。

第 3 章　将移动示例转变为第一人称射击游戏，讲解射线发射和基础 AI。射线发射(向场景发射一条线，并观察相交情况)是所有类型游戏中很有用的操作。

第 4 章　涵盖了美术资源的导入和创建。本章不关注代码，因为每个项目都需要(基本)模型和贴图。

第 5 章　学习如何在 Unity 中创建 2D 益智游戏。尽管 Unity 开始时仅包括 3D 图形，但现在也能很好地支持 2D 图形。

第 6 章　用平台游戏机制扩展了 2D 游戏。特别是，为玩家实现控件、物理和动画。

第 7 章　介绍 Unity 中最新的 GUI 功能。每个游戏都需要 UI，而最新版本的 Unity 为创建 UI 提供了一个改进的系统。

第 8 章　展示如何在 3D 中创建另一种移动游戏，此时从第三人称的视角看到场景。实现第三人称控制将展示一系列 3D 数学操作，学习如何使用带动画的角色。

第 9 章　浏览如何在游戏中实现交互设备和物品。玩家有很多方式操作这些设备，包括直接触摸它们，接触游戏中的触发器，或者是按下控制器的某个按钮。

第 10 章　涵盖了如何与互联网通信。学习如何使用标准互联网技术来发送和接收消息。例如 HTTP 请求，从服务器获取 XML 数据。

第 11 章　介绍如何编写音频功能。Unity 对短音效和长音轨提供了很好的支持，这两种类型的音频对于所有视频游戏都很重要。

第 12 章　将不同章节的碎片整合到一个游戏中。此外，还学习如何编写指向-单击的控件，以及如何保存玩家的进度。

第 13 章　构建最终应用，发布到多个平台，例如桌面、网页和移动，甚至 VR。Unity 与平台无关，允许为每个主流的游戏平台创建游戏。

本书最后还提供了 4 个附录，分别介绍场景导航、外部工具、Blender 和学习资源。

代码约定、要求和下载

本书的所有源代码，不管是代码清单或是片段，都使用等宽字体，以便与周围的文本区别开来。在大多数代码清单中，代码都通过注释指出关键概念，而编号有时用于在文本中提供关于代码的额外信息。代码是经过格式化的，通过合理地增加换行和缩进，以适应本书可用的页面空间。

唯一需要的软件是 Unity，本书使用的是 Unity 2017.1，它是编写本书时的最新版本。某些章节偶尔讨论其他软件，但那些仅作为可选的额外部分，而非核心的学习内容。

警告：

Unity 项目会记住它们在哪个版本的 Unity 中创建，如果尝试在不同版本的 Unity 中打开它们，会显示警告。如果打开本书下载的示例时看到警告，请单击 Continue 并忽略它。

本书的代码清单通常展示了在已有的代码文件中应该添加或修改的内容，除非是首次出现的代码文件，否则不要用后来的清单覆盖整个文件。尽管可以下载书中引用的完整示例项目，但最好输入代码清单中的内容，并观察所引用的示例。可以从 Manning 出版社的网站(www.manning.com/books/unity-in-action-second-edition)和 GitHub(https://github.com/ jhocking/uia-2e)下载示例。也可扫描封底二维码获取本书示例文件。

本 书 论 坛

购买本书， 就可以免费访问由 Manning 出版社运营的私人网页论坛，在该论坛上可以发表关于本书的评论，提问技术问题，并接受来自作者和其他用户的帮助。为了访问论坛，可以进入 https://forums.manning.com/forums/unity-in-action-second-edition，在 https://forums.manning.com/forums/about 中可以了解 Manning 论坛和行为准则。

Manning 对读者的承诺是提供一个场所，让读者之间以及读者与作者之间进行有意义的对话。不承诺作者具体会参与多少分享，作者对论坛的贡献是自愿的(而且是无偿的)。建议试着向作者提出一些具有挑战性的问题，以免令作者兴味索然！只要本书还在印刷，该论坛和之前讨论的档案文件都可以在出版商的网站上找到。

作 者 简 介



Joseph Hocking 是一名软件工程师，专门研究交互式媒体开发。他目前为 InContext Solutions 公司工作，在为 Synapse Games 公司工作期间撰写了本书的第 1 版《Unity5 实战　使用 C#和 Unity 开发多平台游戏》。他还在伊利诺伊大学芝加哥分校、芝加哥艺术学院和哥伦比亚大学芝加哥分校授课。可以访问他的网站 www.newarteest.com。

封面插图声明

本书封面上的插图标题是“Habit of the Master of Ceremonies of the Grand Signior”。Grand Signior 是土耳其帝国苏丹的另一个名称。插图取自 Thomas Jefferys 的 *A Collection of the Dresses of Different Nations, Ancient and Modern (4 volumes)*，这些书在 1757—1772 年于伦敦出版。标题页表明了，这些是手工上色的铜版雕刻，使用阿拉伯树胶增加厚度。Thomas Jefferys(1719—1771)被称为“国王乔治三世的地理学家”。他是一位英国制图师，是当时顶尖的地图供应商，他为政府和其他官方机构雕刻和印刷地图，制作了大量的商业地图和地图集，尤其是北美地图集。作为一名地图绘制师，他的工作激起了人们对他所调查地区的当地服饰习俗的兴趣，这些服饰在这套四卷书集中得到了很好的展示。

在 18 世纪末，兴起了一股风潮，人们开始向往远方，并享受旅行的乐趣。像 Jeffery 画作这样的收藏品是很流行的，为旅行者和向往旅行、但是没能出发的人们介绍异域居民是什么样子。Jeffery 画作藏品的多样性生动描绘了两百多年前各个国度的独特性。从那以后，着装上就发生了变化，而当时各国家、各地区丰富的多样性也渐渐趋同。现在很难区分来自不同大陆的人们。如果从乐观的角度看，我们是把文化和视觉上的多样性作为代价，换来了更丰富的私人生活，或者变化更大、更有趣的知识和技术生活。

在如今这个计算机图书封面大同小异的时代，Manning 出版社以两个世纪前丰富多样的地域生活为基础设计图书封面，令 Jeffery 的画作重新焕发生机，颂扬计算机行业的革新性和首创精神。

目　　录

第 I 部分　起步

第Ⅲ部分 冲刺阶段

第 I 部分

起 步

是时候迈出使用 Unity 的第一步了。如果你一点也不了解 Unity，那也没关系！本部分将首先解释 Unity 是什么，以及使用它编写游戏程序的一些基本原理。接着将讲解一个在 Unity 中开发简单游戏的指南。该指南将讲述一系列游戏开发技术及其大概的工作流程。

下面开始学习第 1 章！

第 1 章

初识 Unity

本章涵盖：

- 是什么使得 Unity 成为一个极佳选择
- 操作 Unity 编辑器
- 在 Unity 中编程
- 比较 C#和 JavaScript

如果你像我一样，曾经有很长一段时间梦想着开发一款视频游戏。但是从玩游戏到实际开发游戏是一个很大的跳跃。这些年出现了很多游戏开发工具，而我们准备讨论的正是这些工具中最现代、最强大的一个。Unity 是一个专业的游戏引擎，它用于创建针对不同平台的视频游戏。它不仅是一个被成千上万经验丰富的开发者使用的开发工具，也是当代游戏开发新手比较容易上手的现代开发工具。直到现在，游戏开发新手在制作游戏时，仍然面临很多巨大的障碍，但 Unity 的出现让这些技能变得更容易学习。

你正在阅读本书，可能是你对计算机技术比较好奇，并且使用其他工具开发过游戏或者构建过其他类型的软件，例如桌面应用或网站。创建一个视频游戏与编写其他类型的软件没有什么根本区别，但也有一定的区别。例如，视频游戏比大多数网站有更多的交互，而且会包含很多不同类型的代码，但制作所用的技术和方法很相似。如果你

已经克服了学习游戏开发道路上的第一道障碍，已经学习了编写软件程序的基本原理，那么下一步就是选择一些游戏开发工具并把编程知识转化到真正的游戏中去。Unity 是开发游戏工作环境的一个极佳选择。

关于术语的提示

这是一本关于 Unity 编程的书，因此它针对的主要是编程人员。尽管很多资源讨论游戏开发和 Unity 中的其他方面，但本书还是把编程放在了首位。

顺便提一下，“开发者”在游戏开发上下文中有着不同的含义：一般情况下，“开发者”和 Web 开发的编程人员是同义词。但在游戏开发中，“开发者”指的是做游戏开发工作的任何人，除了刚刚提及的编程人员，还有其他类型的游戏开发者，如艺术家和设计师。但本书将关注编程部分。

在开始介绍 Unity 前，请先到网站 www.unity3d.com 下载软件。本书使用的是 Unity 2017.1 版本，此版本是编写本书时的最新版本。这个 URL 是 Unity 最初专注于 3D 游戏遗留下来的；它对 3D 游戏的支持依然强大，也能很好地服务于 2D 游戏。本书介绍 2D 和 3D 游戏。实际上，即使在 3D 游戏中演示，许多主题(保存数据、播放音频等)都适用于这两种情况。同时，尽管还有一些付费版本，但基础版本还是完全免费的。本书的所有内容都基于免费版本，不需要 Unity 付费版本。这些版本之间的区别在于商业声明条款。

1.1　为什么 Unity 如此优秀

Unity 是一个专业级的高质量游戏引擎，它用于创建针对多种平台的视频游戏。这个答案相当直接地回答了“什么是 Unity？”这样的问题。然而，这个答案具体意味着什么，为什么 Unity 如此优秀？

1.1.1　Unity 的优势

游戏引擎都提供了充足的特性，这些特性在很多不同的游戏中都有用。因此通过使用引擎，只要加入自定义的美术资源并增加自己游戏玩法的代码，就可以轻松获得那些特性，从而实现一个游戏。Unity 有物理模拟、法线贴图、屏幕空间环境光遮蔽(Screen Space Ambient Occlusion，SSAO)、动态阴影等特性。很多游戏引擎以有诸多特性而自豪，但 Unity 比起其他尖端的游戏开发工具有两个主要优势：提供了非常高效的可视化工作流和多维度的跨平台支持。

可视化工作流是相当独特的设计，它和其他大多数游戏开发环境不同。其他游戏开发工具通常混杂了必定引发争议的不相关部分、需要自己设置集成开发环境(Integrated Development Environment，IDE)的编程类库、构建链和其他陈旧的设计等，

而 Unity 中的开发工作流是通过精心设计的可视化编辑器定位的。这些编辑器用于布局游戏中的场景，将美术资源绑定在一起并对可交互对象进行编码。这些编辑器的美妙之处在于它允许快捷高效地构建专业、高质量的游戏。当需要在视频游戏中使用大量的新技术时，它将提供难以置信的高效工具。

注意 其他大多数带有集中式可视化编辑器的游戏开发工具通常被限制得很死，它们支持的脚本也不灵活，但 Unity 没有这个缺点。尽管 Unity 基本上是通过可视化编辑器创建所有的内容，但这个核心接口还包括了一系列链接到可运行于 Unity 游戏引擎上的自定义代码的项目。通过为项目提供核心接口这种方式很像为诸如 Visual Studio 或 Eclipse 的 IDE 在项目设置中链接类一样。有经验的编程人员应该考虑这个开发环境，不要误以为它只能通过鼠标单击，从而限制了 Unity 的编程能力。

可视化编辑器对于快速迭代以及在原型制作和测试游戏的周期中打磨游戏都非常有益。可以在编辑器中调整物体，甚至是在游戏运行时移动物体。另外，Unity 允许通过编写脚本来自定义编辑器，以在界面上增加一些新特性或菜单。

除了编辑器非凡的生产力优势，另一个主要长处在于 Unity 的工具集提供了高度跨平台的支持。不仅是在部署目标方面跨平台(能部署到 PC、Web、移动设备或游戏主机)，还包括开发工具跨平台(能在 Windows 或 Mac OS 上开发游戏)。这个平台潜力很大，因为 Unity 最开始作为 Mac 独享的软件，后来才移植到 Windows。Unity 的第一个版本在 2005 年发布，现在 Unity 已经更新到了第五个主要版本。最初，Unity 仅支持开发和部署在 Mac 上，但数月后，Unity 已经更新到也能工作在 Windows 上。后续版本的 Unity 中添加了更多部署平台，例如 2006 年添加了可跨平台的 Web 播放器，2008 年添加了对 iPhone 的支持，2010 年添加了支持 Android，甚至支持更多游戏主机，比如 Xbox 和 PlayStation。最近它们已经添加了到 WebGL 的部署，WebGL 是一个用于 Web 浏览器 3D 图形的新框架，甚至支持 VR 平台，如 Oculus Rift 和 Vive。一些游戏引擎也支持和 Unity 一样多的部署目标，但是没有一个能像 Unity 那样让部署到多平台的工作变得如此简单。

除了这些主要的优点外，第三条微妙的优点是 Unity 使用模块化组件系统构造游戏对象。在一个组件系统中，“组件”是一个混合搭配功能的包，对象由一系列组件构建，而不是由层级严格的类构建。换句话说，组件系统是和面向对象编程不同的方法(它更灵活)，游戏对象是通过组合的方式而不是继承的方式构建的。图 1-1 演示了继承和组件系统的对比。

在组件系统中，物体存在于一个扁平的层级结构中，不同的物体有不同的组件集合，而不像继承结构，不同物体处于树的完全不同的分支中。这种设计加快原型的开发，因为当物体改变时，可以快速混合搭配不同组件而不必重构继承链。

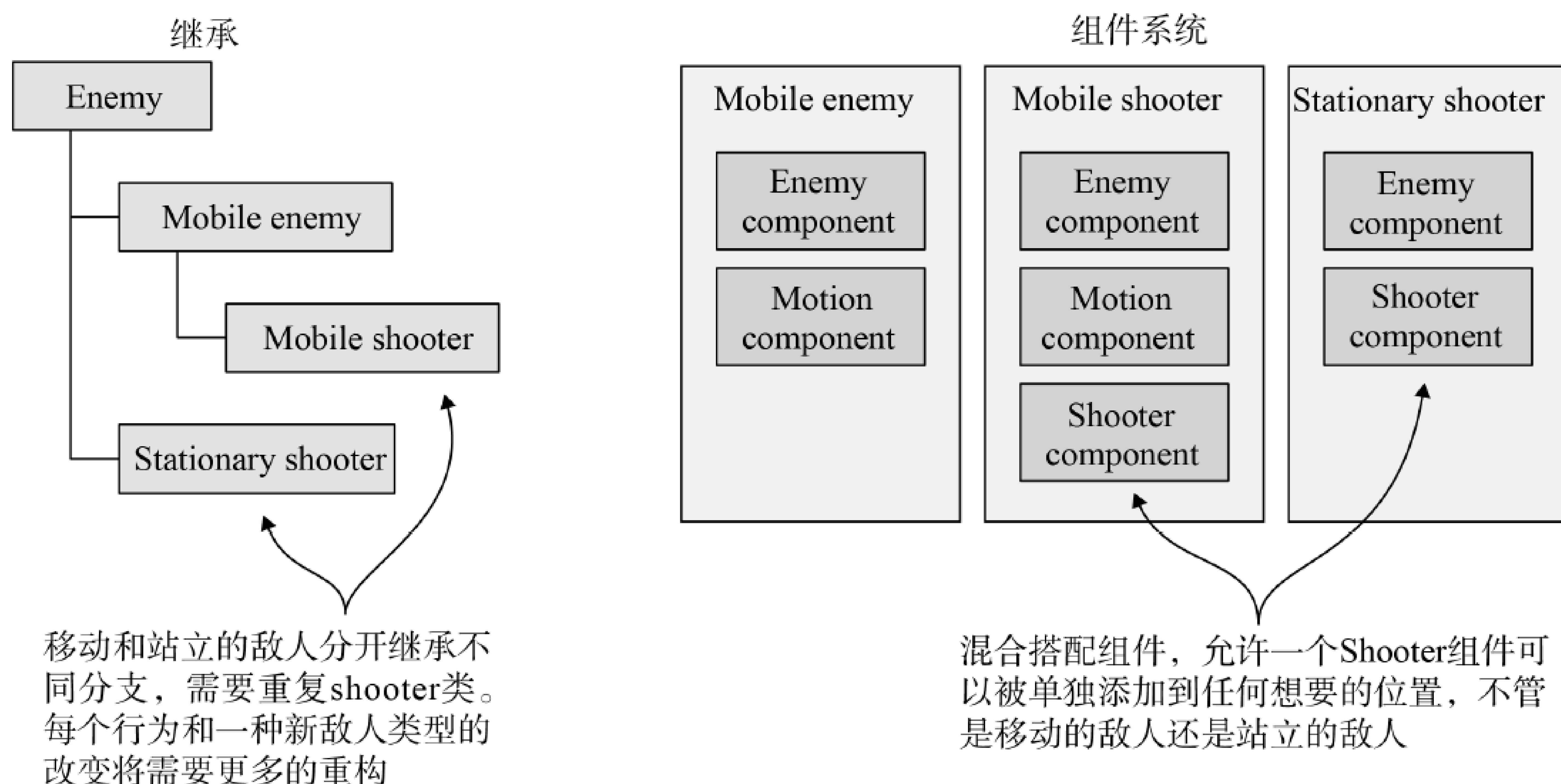


图 1-1　继承与组件系统

当没有组件系统时，可以编写代码实现自定义的组件系统，但是 Unity 已经有一个健壮的组件系统，这个系统甚至与可视化编辑器无缝地集成在一起。不仅能通过代码维护组件，还能使用可视化编辑器附加和移除组件。另外，可以通过组合构建对象，也可以在代码中选择使用继承，包括所有基于继承的最佳设计模式。

1.1.2　要意识到的缺点

Unity 有很多优点，这让它成为开发游戏的好选择，我们也极力推荐它，但如果不提它的缺点就有点失职了。实际上，混合使用可视化编辑器和复杂的代码，尽管和 Unity 的组件系统能高效地组合在一起，但它也不简单。在复杂场景中，你会搞不清楚场景中的哪个物体附加了指定组件。Unity 提供了搜索功能用于查找附加的脚本，但该搜索是很粗糙的；有时你还会遇到一些情况，为了找出脚本链接，需要手动检查场景中的物体。这些情况虽然不会经常出现，但是一旦出现，会令人沮丧。

第二个缺点会让有经验的编程人员吃惊且失望，即 Unity 不支持链接外部代码库。需要的库必须手动复制到每个使用它们的项目中，而不是引用一个中心共享的位置。对类库缺少中心位置，在多个项目中共享类库就会变得笨拙。这个缺点能通过巧妙使用版本控制系统来绕开，但 Unity 不支持外部代码库的使用。

注意　难以使用版本控制系统(例如 Subversion、Git 和 Mercurial)在过去是 Unity 的一个致命弱点，但现在 Unity 的一些版本已经可以使用它们。一些过时的资源提到，Unity 不能使用版本控制，但是一些新资源会说明，项目中的哪些文件和文件夹需要放在资源库里，哪些不用。为使用版本控制系统，请阅读 Unity 的文档(http://mng.bz/BbhD)或查看 GitHub(http://mng.bz/g7nl)维护的.gitignore 文件。

第三个缺点是必须使用预设(prefab)。预设是 Unity 特有的一个概念，将在第 3 章中解释，现在只需要知道预设是可视化定义交互对象的一种灵活方式。预设很强大，只属于 Unity(它和 Unity 的组件系统绑定在一起)，但预设的编辑流程却十分粗糙。由于预设极其有用，且作为 Unity 的主要部分进行工作，因此希望在未来的 Unity 版本中能改善对预设进行编辑的工作流。

1.1.3　使用 Unity 构建的游戏示例

前面已经介绍了 Unity 的优缺点，但仍然需要确信 Unity 中的开发工具是最好的选择。请访问 Unity 图片库 http://unity3d.com/showcase/gallery，看一看使用 Unity 开发的游戏和仿真程序。这个列表包含成百上千个游戏和仿真程序，并且在不断更新中。本节仅介绍一些类型和部署平台的游戏示例。

1. 桌面(Windows、Mac、Linux)

Unity 编辑器运行在同一个平台上，并通常部署到诸如 Windows 或 Mac 这样比较常见的目标平台上。下面给出一些不同类型的桌面游戏示例：

- Guns of Icarus Alliance(见图 1-2)，一个由 Muse Games 开发的第一人称射击游戏。



图 1-2　Guns of Icarus Alliance 游戏

- Gone Home(见图 1-3)，一个由 Fullbright Company 开发的冒险游戏。



图 1-3　Gone Home 游戏

2. 移动平台(iOS 和 Android)

Unity 能将游戏部署到移动平台上，比如 iOS(iPhone 和 iPad)和 Android(手机和平板电脑)。下面是一些不同类型的移动平台的游戏：

- Lara Croft GO(见图 1-4)，由 Square Enix 开发的 3D 拼图游戏。



图 1-4　Lara Croft GO 游戏

- INKS(见图 1-5)，由 State of Play 开发的 2D 拼图游戏。

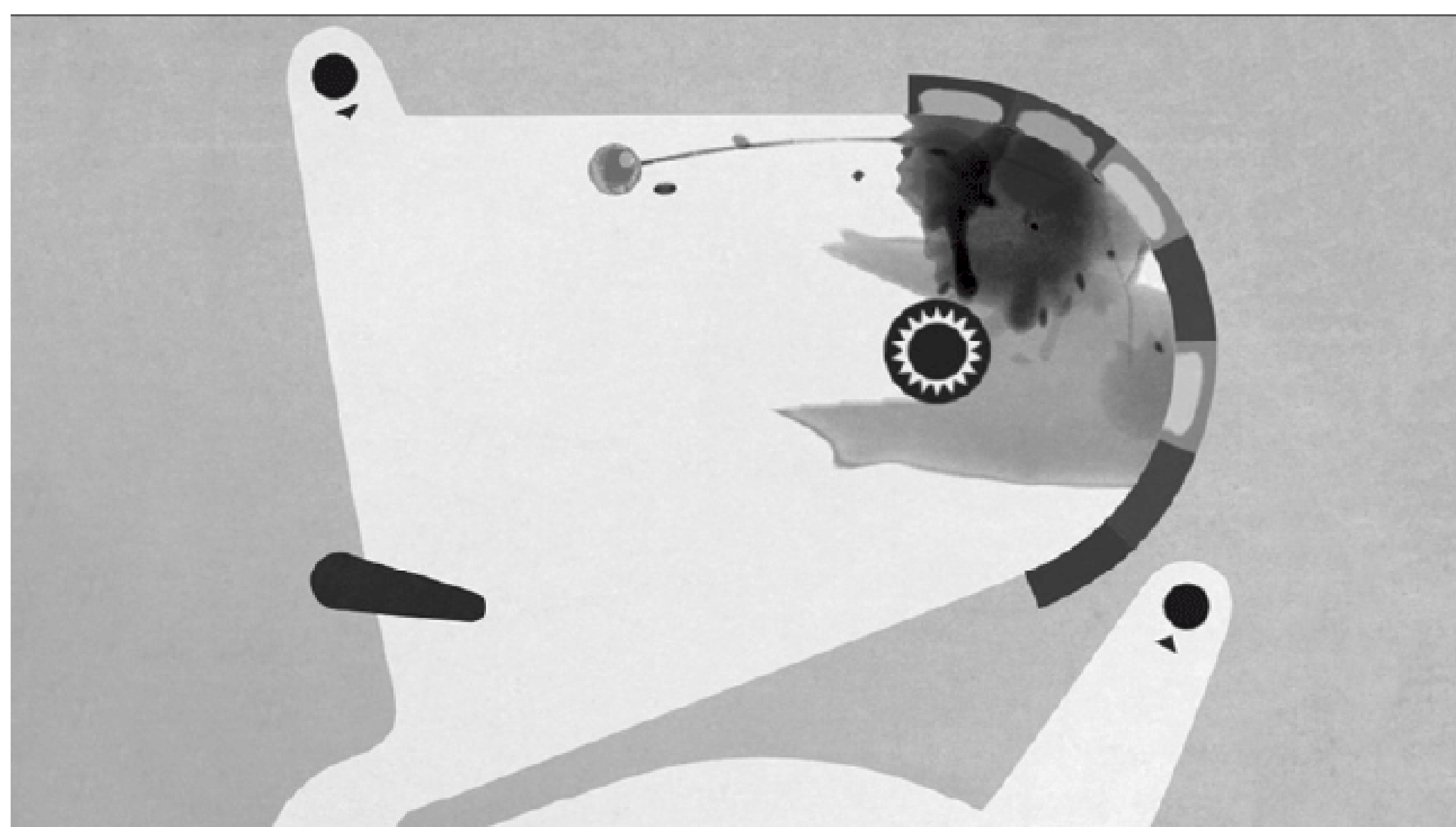


图 1-5　INKS 游戏

- Tyrant Unleashed(见图 1-6)，由 Synapse Games 开发的收集类卡牌游戏。

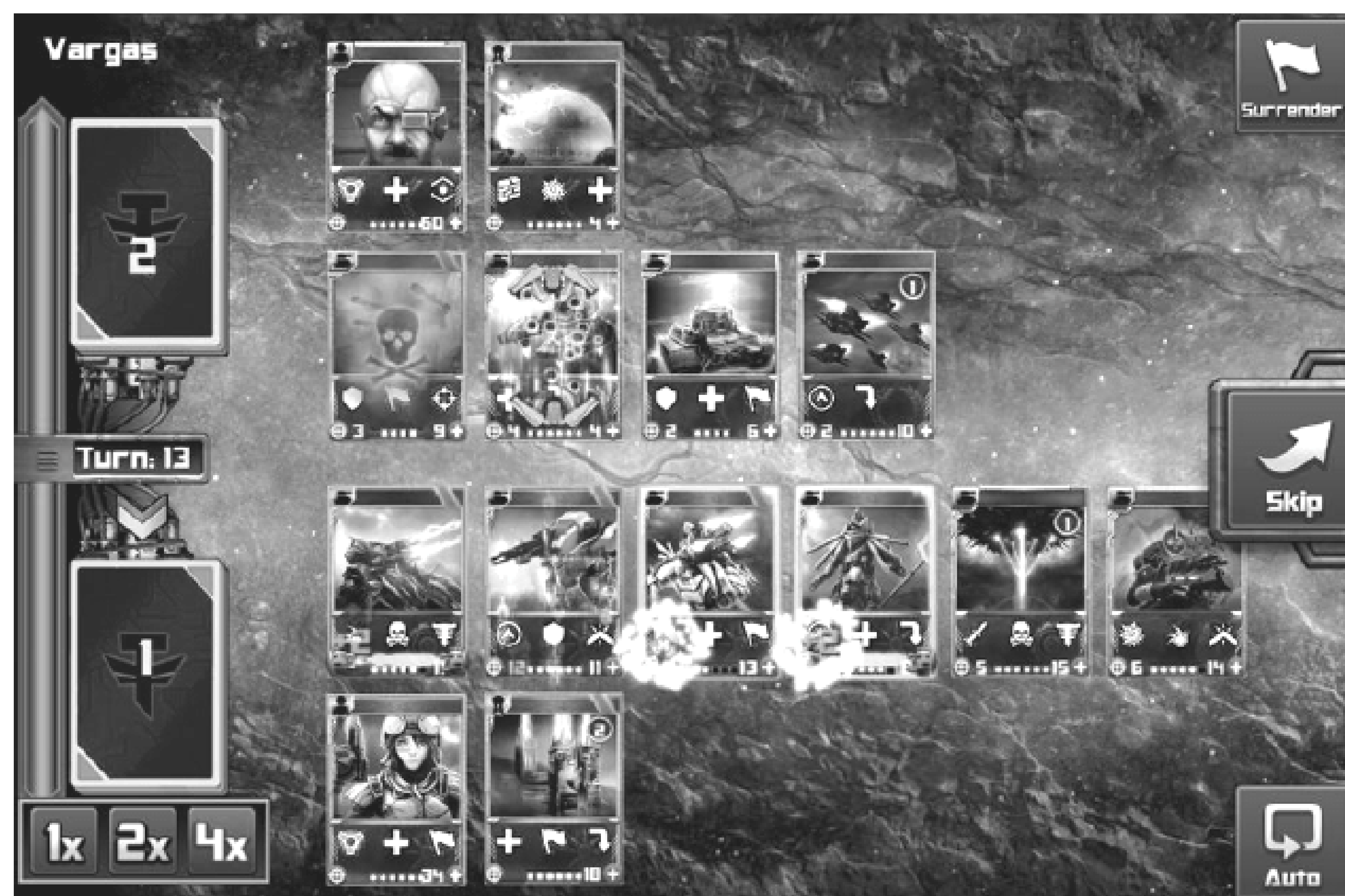


图 1-6　Tyrant Unleashed 游戏

3. 主机(PLAYSTATION，XBOX，SWITCH)

Unity 能将游戏部署到游戏主机，但开发者依然需要从 Sony、Microsoft 或 Nintendo 获得许可。因为有这种需求和 Unity 容易跨平台部署的特性，主机游戏通常在桌面计算机上也可以看到。以下是一些不同类型的主机游戏的示例：

- Yooka-Laylee(见图1-7)，由Playtonic Games开发的3D Platformer游戏。



图 1-7　Yooka-Laylee 游戏

- Shadow Tactics(见图 1-8)，由 Mimimi Productions 开发的隐形游戏。



图 1-8　Shadow Tactics 游戏

从这些示例中可以看到，Unity 的强大功能肯定可以应用到商业品质的游戏中。虽然 Unity 超越其他游戏开发工具的优势明显，但新手可能会误解编程在开发流程中的存在。Unity 通常表现得就像是一个简单的特性列表，它不需要编程，其实这是错误的观点，它并没有告诉人们制作一款商业游戏需要什么。让大家误以为可以通过单击，使用现有的组件，甚至不需要编程人员就能制作一个精良的原型。严格来讲，将一个有趣的原型变成可发布的精品游戏，是必须要编程的。

1.2　如何使用 Unity

上一节讨论了很多通过 Unity 的可视化编辑器提高生产率的问题，因此下面介绍 Unity 的界面以及如何操作它。如果你尚未下载 Unity，请从 www.unity3d.com 下载程序，并在计算机上安装(在安装程序中请确认选中了 Example Project)。在安装后，运行 Unity，开始浏览界面。

为了通过示例了解该界面，请打开所包含的示例项目。刚安装好的 Unity 会自动打开示例项目，也可以选择 File | Open Project 来手动打开它。示例项目安装在共享的用户目录中，例如，在 Windows 上是在 C:\Users\Public\Documents\UnityProjects\中；在 Mac OS 上是在 Users/Shared/Unity/中。还可能需要打开示例场景，双击 Car 场景文件(图 1-9 中高亮显示的文件，场景文件图标是 Unity 的立方体)，这个文件可以在编辑器底部的文件浏览器的 SampleScenes/Scenes/中找到。图 1-9 显示了该界面。

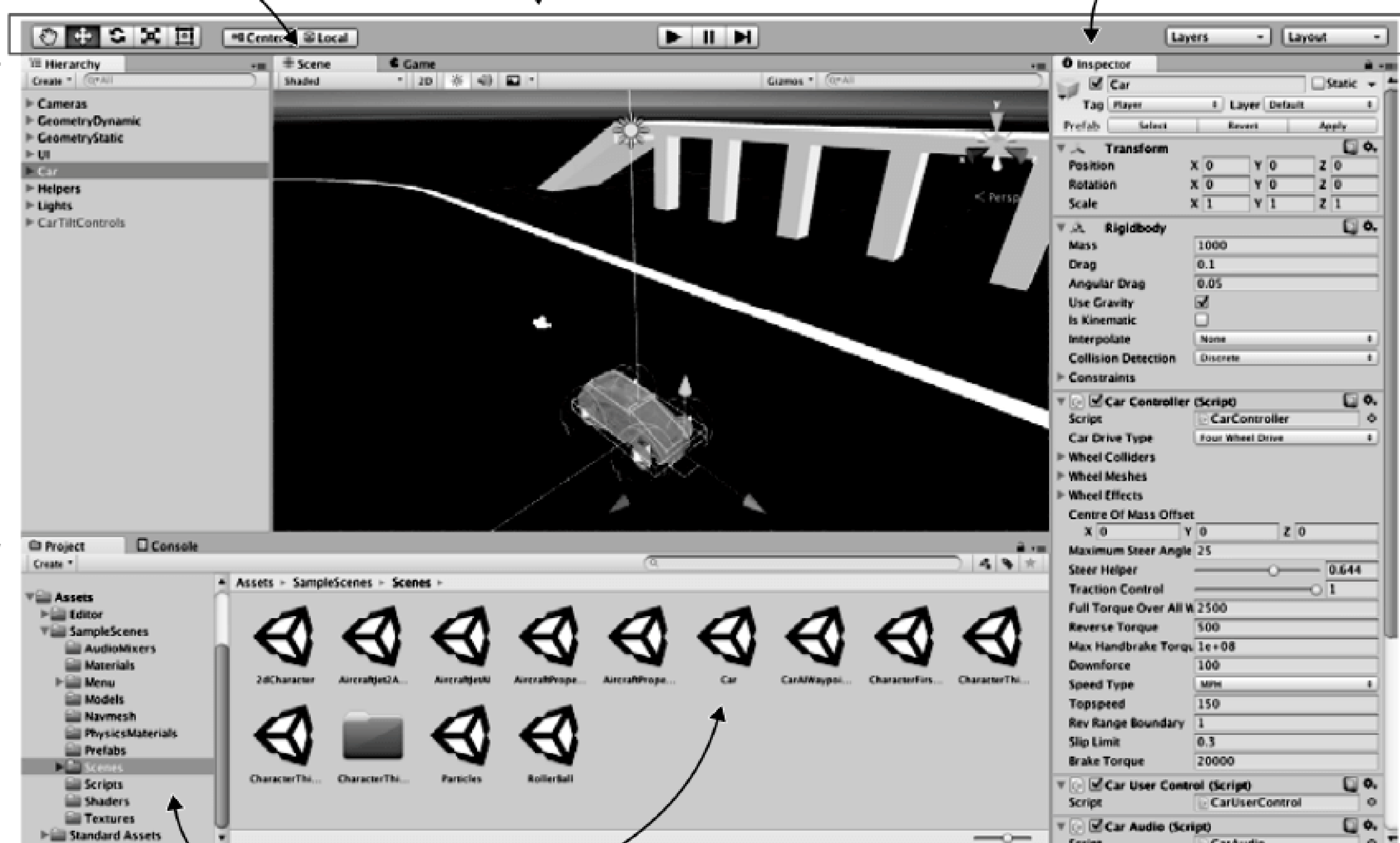


图 1-9　Unity 中的部分界面

Unity 中的界面分为不同的部分：Scene 标签、Game 标签、工具栏、Hierarchy 标签、Inspector 标签、Projector 标签和 Console 标签。每个部分都有不同的用途，它们在构建游戏的生命周期中都很重要：

- 可以在 Project 标签中浏览所有文件。
- 使用 Scene 标签，可以在浏览 3D 场景的同时将对象放置进去。

- 工具栏中的控件用于处理场景。
- 可以在 Hierarchy 标签中拖放对象的关系。
- Inspector 列出了所选对象的信息，包括链接的代码。
- 可以在 Game 视图中测试游戏，并在 Console 标签中查看错误输出。

这是 Unity 中的默认布局，所有不同的视图都有标签，而且可以移动它们或修改它们的尺寸，将其停靠在屏幕中不同的位置。后面将介绍如何自定义布局，不过现在默认布局是理解所有视图用途的最佳方式。

1.2.1　Scene 视图、Game 视图和工具栏

界面中最突出的部分是中间的 Scene 视图。这是观察游戏世界和移动对象的场所。网格对象也会出现在场景中(后面给出它的定义)。也能在场景中看到其他对象，它们由不同的图标和彩色线条表示：摄像机、灯光、声源、碰撞区域等。注意在这个视图看到的东西和运行游戏时看到的不一样。可以在 Scene 视图中到处浏览，而不会受到 Game 视图的约束。

定义　网格对象是 3D 空间中的可视对象。3D 中的可视对象是由很多连接的线和图形构成的，因此世界是由网格构成的。

Game 视图不是屏幕中独立的部分，而是另一个标签，它位于 Scene 视图的右侧(在视图左上角可以找到该标签)。在界面上的有些地方可能有多个这样的标签，如果单击一个不同的标签，视图会被新激活的标签替换。当游戏运行时，这个视图会变成 Game 视图。在每次运行游戏时不需要手动切换标签，因为当游戏启动时，视图会自动切换为 Game 视图。

提示　运行游戏时，可以切换回 Scene 视图，这样就能在运行的场景中检查对象。运行游戏时，这个功能非常适合于查看什么对象在执行什么操作，它是一个很有用的调试工具，而这正是大多数引擎所不具备的。

所谓运行游戏，就是简单地单击 Scene 视图上方的 Play 按钮。界面的顶部是工具栏，Play 按钮正位于中间。图 1-10 为了展示顶部的工具栏，把整个编辑器界面分开了，Scene/Game 标签位于工具栏的下面。

工具栏左边的按钮用于场景导航和变换对象——如何浏览场景和移动对象。建议花一些时间练习浏览场景和移动对象，因为它们是 Unity 可视化编辑器中最重要的两个操作(因为它们非常重要，所以后面有两小节专门介绍它们)。工具栏的右侧是布局和图层的下拉菜单。如前所述，Unity 界面的布局很灵活，Layouts 菜单允许在布局之间来回切换。Layers 菜单具有高级功能，现在可以先忽略它(后续章节中将介绍)。

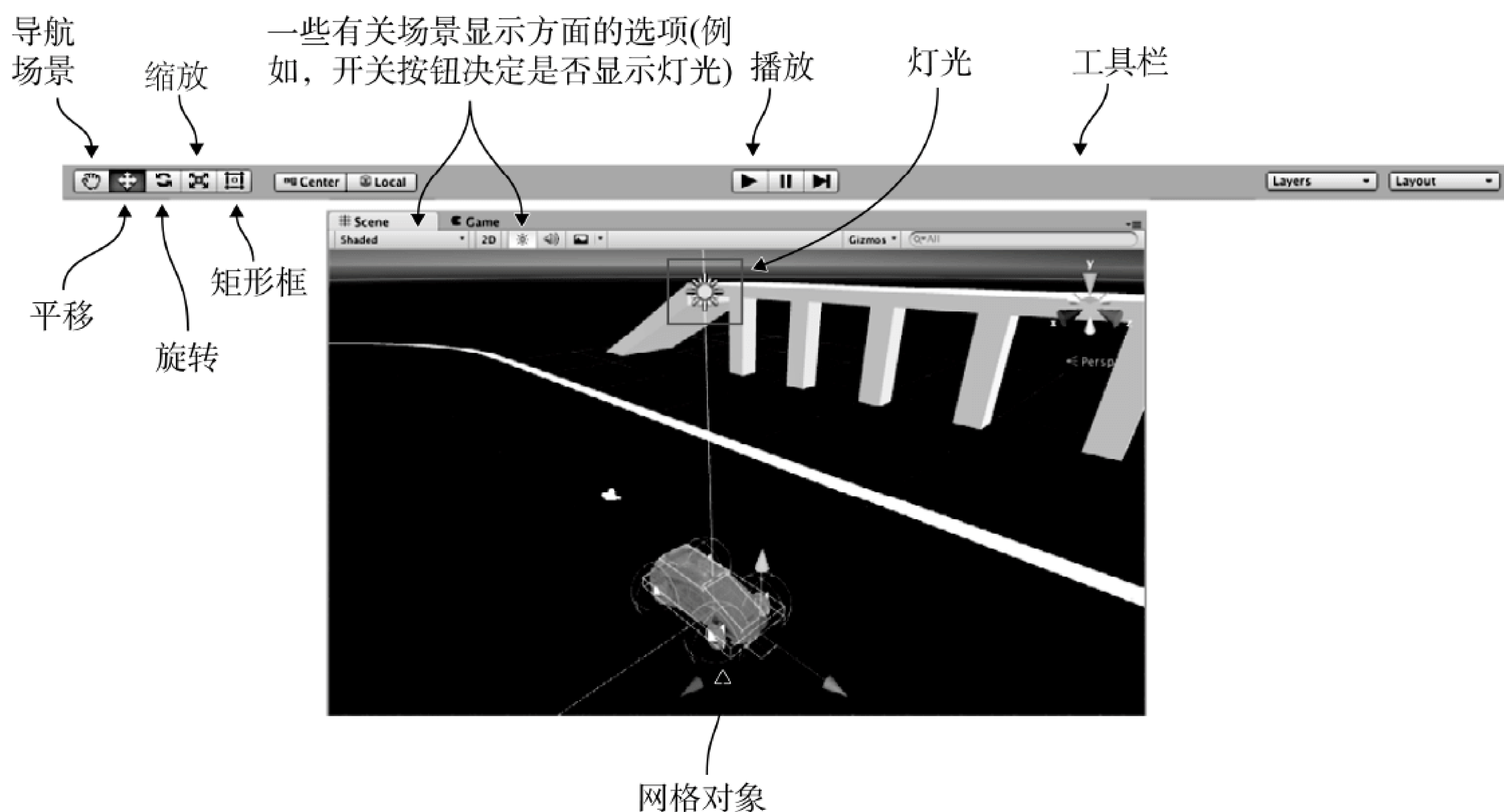


图 1-10　编辑器屏幕截图用于显示工具栏、场景和游戏

1.2.2　使用鼠标和键盘

场景导航主要使用鼠标，通过一些修饰键来修改当前鼠标的行为。这三个主要的导航菜单项是移动(Move)、盘旋(Orbit)和变焦(Zoom)。具体的鼠标移动操作在本书的附录 A 中给出。这三种不同的移动是指按住一些 Alt(在 Mac 上是 Option)和 Ctrl 的组合键时的单击和拖动操作。花几分钟在场景中移动，了解移动、盘旋和变焦的作用。

提示　虽然 Unity 能使用带一个或两个按钮的鼠标，但强烈建议使用带三个按钮的鼠标(带三个按钮的鼠标在 Mac OS X 中也工作得很好)。

通过三个主要的菜单项可以完成对象的变换，而三个场景导航操作也类似于三个变换操作：平移(Translate)、旋转(Rotate)和缩放(Scale)。图 1-11 演示了如何变换一个立方体。

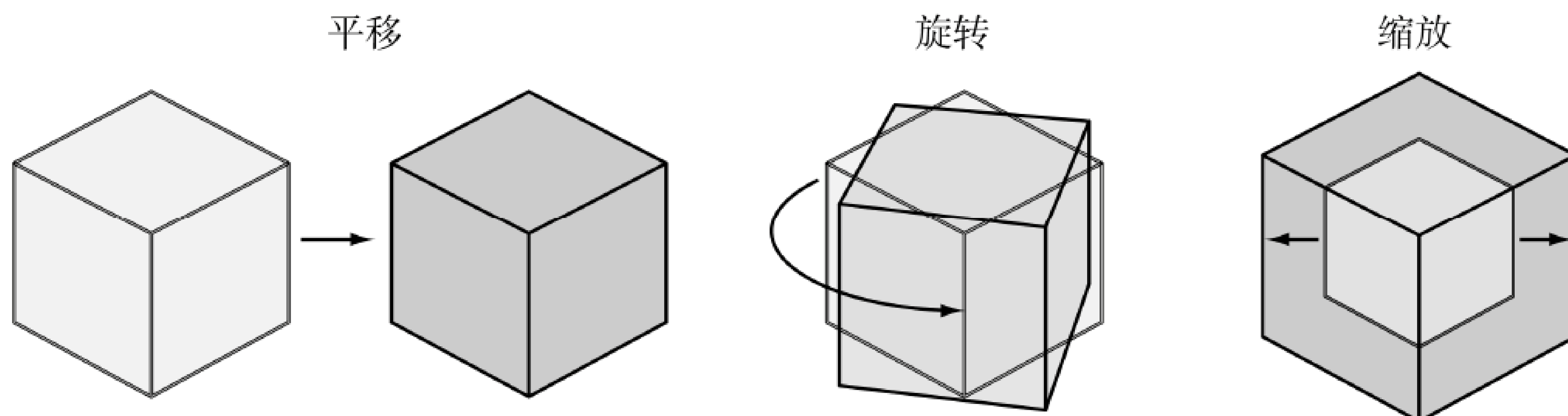


图 1-11　应用三种变换：平移、旋转和缩放(较浅的线图是对象在变换前的状态)

选中场景中的一个对象后，就能移动(数学上精确的技术术语是平移)、旋转或缩放它。在场景导航菜单项中，Move 是平移摄像机，Orbit 是旋转摄像机，Zoom 是缩

放摄像机。除了使用工具栏的按钮之外，按下键盘的 W、E 或 R 键还能切换这些功能。当激活一个变换操作时，会有一系列彩色箭头或圆圈围绕着场景中的对象，这是 Transform 的 gizmo，可以单击或拖动 gizmo 来进行变换。

在变换按钮旁边还有第四个工具，即 Rect 工具，它用于 2D 图形。该工具组合了移动、旋转和缩放功能。这些操作在 3D 中是分开的，但在 2D 中是组合在一起的，因为少了一个需要考虑的维度。Unity 有许多其他键盘快捷键来加速不同的任务。可以参考附录 A 学习它们。下面介绍界面部分剩下的内容！

1.2.3　Hierarchy 视图和 Inspector 面板

在屏幕边缘，左边是 Hierarchy 标签，右边是 Inspector 标签(见图 1-12)。Hierarchy 是一个列表视图，列出了场景中每个对象的名称，名字是否嵌套取决于它们在场景中链接的层次。基本上，选择对象的方式是通过名称，而不是在场景中逐个查找。Hierarchy 将对象连成一组，看起来它们就像文件夹，并且允许一起移动整个组。

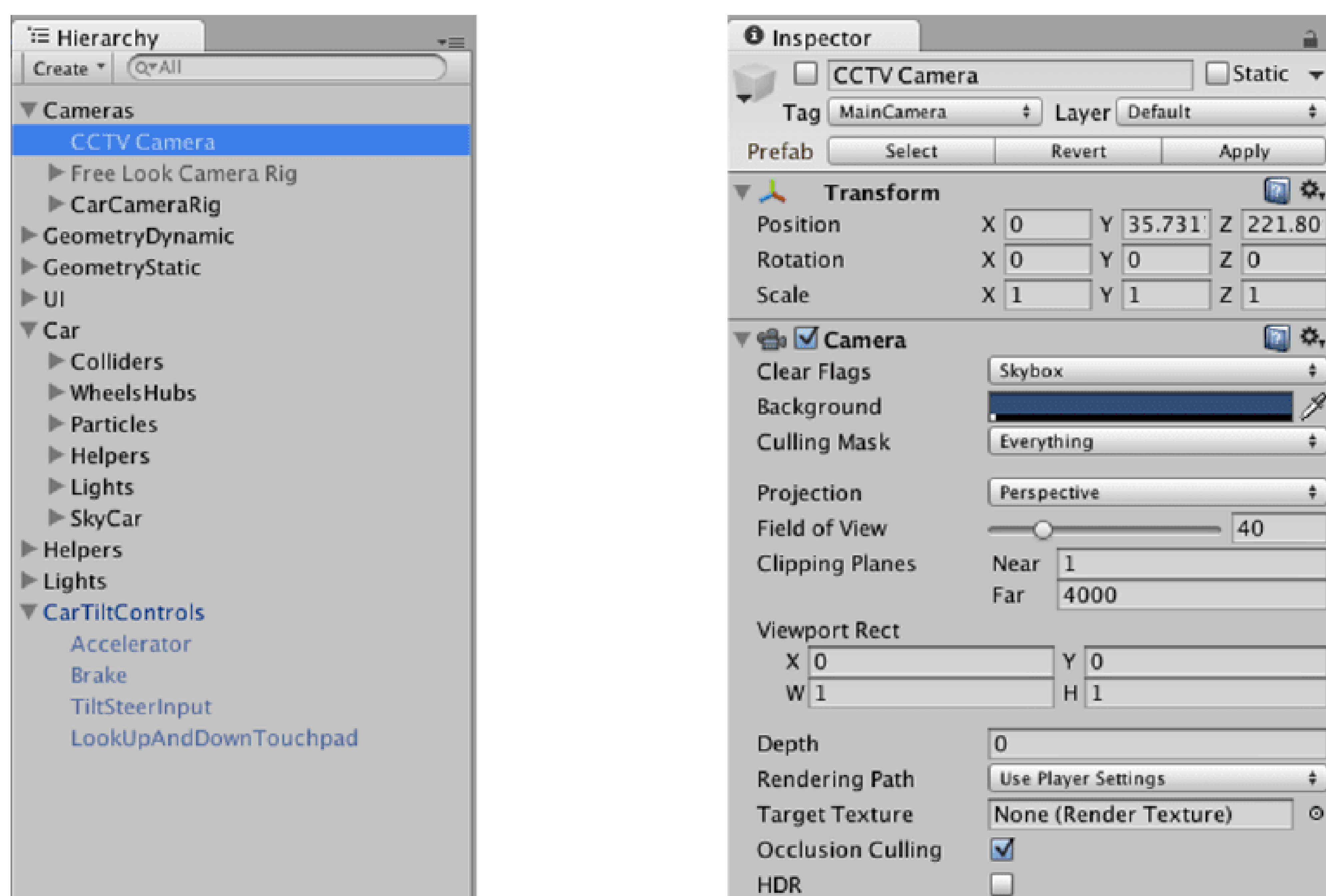


图 1-12　编辑器屏幕截图，用于展示 Hierarchy 标签和 Inspector 标签

Inspector 显示当前所选对象的信息。选择某个对象后，该对象的信息便会填充 Inspector。所显示的信息大部分是组件列表，甚至可以从对象上添加或移除组件。所有游戏对象至少有一个组件，即 Transform，所以在 Inspector 中至少可以看到位置和旋转的信息。很多时候，对象会在 Inspector 中列出很多组件，包括它关联的脚本。

1.2.4　Project 和 Console 标签

屏幕底部是 Project 和 Console 标签(见图 1-13)。与 Scene 和 Game 视图一样，这两个标签不是屏幕中两个独立的部分，可以通过标签在它们之间切换。Project 显示项目中所有的资源(美术、代码等)。具体而言，视图左边是项目中目录的列表；当选择

一个目录时，视图右边会显示该目录中独有的文件。Project 中列出的目录类似于 Hierarchy 中的列表视图，但 Hierarchy 只显示场景中的对象，Project 则显示不包含在特定场景中的文件(包括场景文件——当保存场景时，它会显示在 Project 中)。

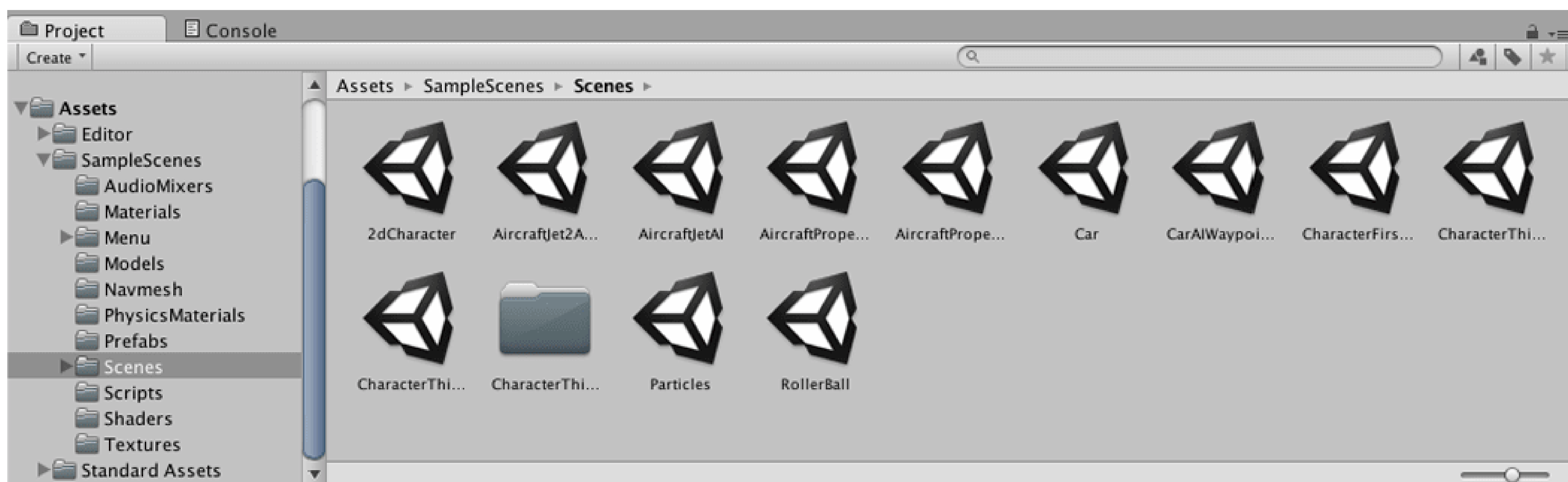


图 1-13　编辑器屏幕截图，用于展示 Project 和 Console 标签

提示 Project 视图镜像了磁盘上的 Assets 目录，但通常不应该直接从 Assets 文件夹中移动或删除文件。如果在 Project 视图中执行这些操作，Unity 会与那些文件夹保持同步。

代码将消息显示在 Console 中。这种消息是专门输出来调试程序的，但是当编写的脚本出现问题时，Unity 也会在 Console 中发出错误消息。

1.3　开始使用 Unity 编程

现在观察 Unity 中编程工作的流程。虽然可以在可视化编辑器中布局美术资源，但需要编写代码来控制它们并让游戏变得可以交互。Unity 支持一些编程语言，特别是 JavaScript 和 C#。它们各有优缺点，本书建议使用 C#。

为什么选择 C#而不是 JavaScript?

本书所有的代码都使用 C#编写，因为相对于 JavaScript，C#有很多的优点，缺点较少，特别是对于专业开发者而言。

C#的一个优点在于它是强类型语言，而 JavaScript 不是。现在，在有经验的编程人员之间存在关于动态类型更适合开发的争议，特别是 Web 开发，但针对特定的游戏平台(比如 iOS)，通常最好甚至必须使用静态类型。Unity 甚至增加了指令#pragma strict 来强制 JavaScript 使用静态类型。尽管技术上可以这么做，但它破坏了 JavaScript 工作的基本原理，而如果想这么做，更好的方式是使用原本就是强类型的语言。

这仅是 Unity 中的 JavaScript 不同于其他地方的 JavaScript 的一个示例。Unity 中的 JavaScript 肯定类似于 Web 浏览器的 JavaScript，但在每个上下文中语言的工作方式还是有很多区别的。很多开发者将 Unity 中的 JavaScript 称为 UnityScript，这个名

字表明了它们之间的相似性，又和 JavaScript 做了区分。这种“相似但不同”的状态给编程人员带来了一些问题，既要学习 Unity 外的 JavaScript 知识，又要在 Unity 中应用学到的编程知识。

同时，由于这些原因，Unity 正在删除 JavaScript/UnityScript 支持。如博客 http://mng.bz/B9au 所述，这种支持正逐渐被淘汰。

下面编写和运行一些代码，尝试第一个示例。启动 Unity，创建一个新项目。在 Unity 的开始窗口中选择 New，如果 Unity 已经在运行，就选择 File | New Project，打开 New Project 窗口。输入项目名称，保留默认 3D 设置(后续章节会介绍 2D)，然后选择一个位置，保存这个项目。Unity 项目只是一个目录，其中包含了不同的资源和设置文件，所以可以在计算机上的任何位置保存项目。单击 Create Project，之后 Unity 会暂时消失，此时它在建立项目目录。

警告 Unity 项目会记住它们是在哪个 Unity 版本中创建的，如果尝试在不同版本的 Unity 中打开它们，将会出现警告。有时这些警告无关紧要(例如，当打开本书下载的示例时，出现警告时忽略它即可)，但有时，需要在打开项目之前备份项目。

同样，打开下载的示例时，Unity 可能会发出以下警告：重新构建库，因为找不到资源数据库！这是指项目的 Library 文件夹。该文件夹包含 Unity 生成并在工作时使用的文件，但不需要分发这些文件。

当 Unity 重新出现时，会看到一个空的项目。下一步，将讨论如何在 Unity 中执行程序。

1.3.1　代码在 Unity 中运行：脚本组件

Unity 中所有代码的执行都从链接到场景对象的代码文件开始。前述的组件系统最终是由代码文件构成的；游戏对象是由组件集合构建而成的，而这些集合可以包含要执行的脚本。

注意 Unity 将代码文件当成脚本，使用“脚本”这个定义，这和运行在浏览器中的 JavaScript 大致一样：代码运行在 Unity 游戏引擎中，不像编译好的代码运行在它自己的可执行环境中。但不要混淆这些概念，因为很多人对这个词的定义是不同的；例如“脚本”通常也指短小、完整的实用程序。Unity 中的脚本更像是独立的 OOP 类，附加到场景对象上的脚本是对象的实例。

Unity 中的脚本是指组件——并非所有脚本，注意，只有从 MonoBehaviour 继承的脚本才指组件，MonoBehaviour 是脚本组件的基类。MonoBehaviour 定义一些看不

见的基础工作，使组件附加到游戏对象上，而继承它，会提供一系列自动运行的方法(见代码清单 1.1)，可以重写它们。这些方法包括 Start()，当对象变成激活状态时(通常是加载带有该对象的关卡)会调用它一次。还包括 Update()，它会在每帧调用。因此，把代码放到这些预定义的方法中时，代码就会运行。

定义 帧是游戏循环代码中的一个周期。几乎所有的视频游戏(不仅 Unity，包括大多数视频游戏)都是围绕一个核心游戏循环建立的。当游戏运行时，代码会在每个周期中执行。每个周期都包括了绘制屏幕，因此命名为帧(电影也由一系列的帧组成)。

代码清单 1.1　基本脚本组件的代码模板

```
using UnityEngine;                              ← 包含用于 Unity 和 Mono 类的
using System.Collections;                         名称空间
using System.Collections.Generic;

public class HelloWorld : MonoBehaviour {       ← 用于继承的语法
    void Start() {
        // do something once                    ← 把运行一次的代码放在这里
    }

    void Update() {
        // do something every frame             ← 把每帧运行的代码放在这里
    }
}
```

这是创建新 C#脚本时文件包含的内容：定义一个合法的 Unity 组件需要的最少模板代码。Unity 有一个脚本模板隐藏在应用内部，当创建新脚本时，它复制该模板，并重命名新类，使之匹配文件名(本示例中的名称为 HelloWorld.cs)。这个脚本还包含空的 Start()和 Update()方法，因为我们大都在这两个方法中调用自己的代码。

为了创建脚本，从 Create 菜单中选择 C# Script，Create 菜单在 Assets 菜单中(注意 Assets 和 GameObjects 都有 Create 的列表，但它们是不同的菜单)，或者在 Project 视图的右击菜单中。输入新脚本的名称，例如 HelloWorld。如本章后面所述(见图 1-15)，单击并拖动这个脚本文件到场景中的对象上。双击这个脚本，它就会在另一个程序中自动打开，进行编辑，如下所述。

1.3.2　使用 MonoDevelop，跨平台的 IDE

准确而言，编程并不是在 Unity 中进行的，代码是 Unity 指向的独立文件。脚本文件能在 Unity 内创建，但仍然需要使用文本编辑器或 IDE 在初始为空的文件中编写所有代码。Unity 绑定了 MonoDevelop，它是一个开源、跨平台、编写 C#的 IDE(见图 1-14)。可以访问 www.monodevelop.com 进一步学习这个软件，但这里使用的版本是和 Unity

绑定的，而不是从网站下载的版本，因为基础软件进行了一些修改，以更好地集成到 Unity 中。

不要在MonoDevelop中单击Run按钮；在Unity中单击Play来运行代码

默认情况下，Document Outline (文档大纲)可能不显示。选择 View | Pads并将该标签拖放到想要的位置

脚本文件在主视图区域作为标签打开。多个脚本文件能同时打开

Solution(解决方案)视图显示项目中所有的脚本文件

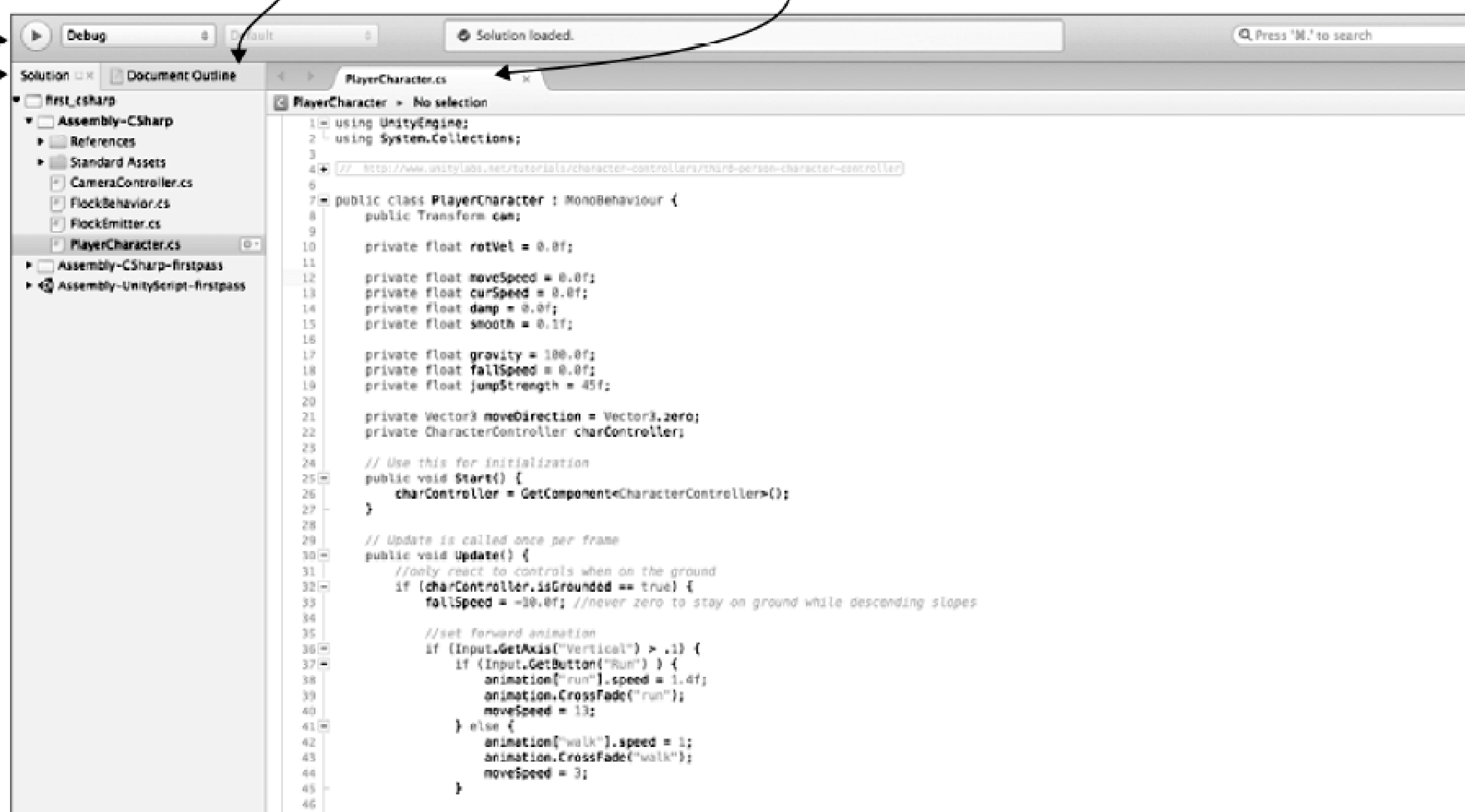


图 1-14　MonoDevelop 中的界面

注意　MonoDevelop 把文件组织成组，称为解决方案。Unity 自动生成一个解决方案，它包含所有脚本文件，因此通常不必关心它们。

C#最开始是 Microsoft 的产品，能使用 Visual Studio 在 Unity 中编程吗？答案是可以。支持工具可以使用 Visual Studio 给 Unity 编程(特别是调试和断点能正常工作)。要查看此支持工具是否已经安装，请检查 Debug 菜单中的 Attach Unity Debugger 选项。如果没有安装，只需要运行 Visual Studio Installer，来修改安装，并查找 Unity 游戏开发模块。

本书主要使用 MonoDevelop，但如果已经使用了 Visual Studio 来编程，则可以继续使用它，学习本书的内容不会产生任何问题(该章之后不会再讨论 IDE)。但是将工作流限制在 Windows 上，会和使用 Unity 的最大优点背道而驰。虽然 C#最初是一个 Microsoft 产品，只能在 Windows 和.NET Framework 中工作，但是现在它已经成为开源标准的语言，而且有一个有意义的跨平台框架：Mono。Unity 使用 Mono 支撑它的编程，而使用 MonoDevelop 允许使整个项目的开发工作流保持跨平台。

警告　MonoDevelop 是与 Unity 2017.1 版本捆绑在一起的 IDE，但如 Unity 博客 http://mng .bz/9HR8 所述，这将在 Unity 2018.1 版本中改变。

时刻记住，尽管代码是使用 Visual Studio 或 MonoDevelop 编写的，但代码不在 Visual Studio 或 MonoDevelop 中运行。IDE 只是一个强大的文本编辑器，只有在 Unity

中单击 Play 按钮，代码才会运行。

1.3.3　打印到控制台：Hello World！

至此，项目中已经有了一个空脚本，但场景中还需要一个附加这个脚本的对象。图 1-1 描述了组件系统的工作原理；脚本是一个组件，因此需要设置为对象上的一个组件。

选择 GameObject | Create Empty，在 Hierarchy 列表中将出现一个空的 GameObject。现在从 Project 视图将脚本拖到 Hierarchy 视图，并放在那个空的 GameObject 上。如图 1-15 所示，Unity 将高亮验证可以放置脚本的地方，在 GameObject 上释放它，使脚本附加到对象上。为验证脚本是否已附加到对象上，选择该对象并查看 Inspector 视图。其中会列出两个组件：一个是 Transform 组件，它是基础位置/旋转/缩放组件，所有对象都包含该组件，而且不能移除它；另一个组件就是脚本。



图 1-15　将脚本链接到 GameObject 上

注意　将对象从一个地方拖到其他对象上并释放的操作是很常见的。Unity 中很多不同的链接都是通过将对象拖到其他对象之上来创建的，不只是将脚本关联到对象上。

脚本链接到对象后，其结果将如图 1-16 所示，脚本像组件那样显示在 Inspector 中。播放场景时，脚本就会执行，不过现在还不会发生任何事情，因为还没编写任何代码。下面接着介绍下一步！

打开 MonoDevelop 中的脚本，回到代码清单 1.1。当学习新的编程语言环境时，最经典的做法是输出文本“Hello World!”，将这行文本添加到 Start()方法中，如代码清单 1.2 所示。

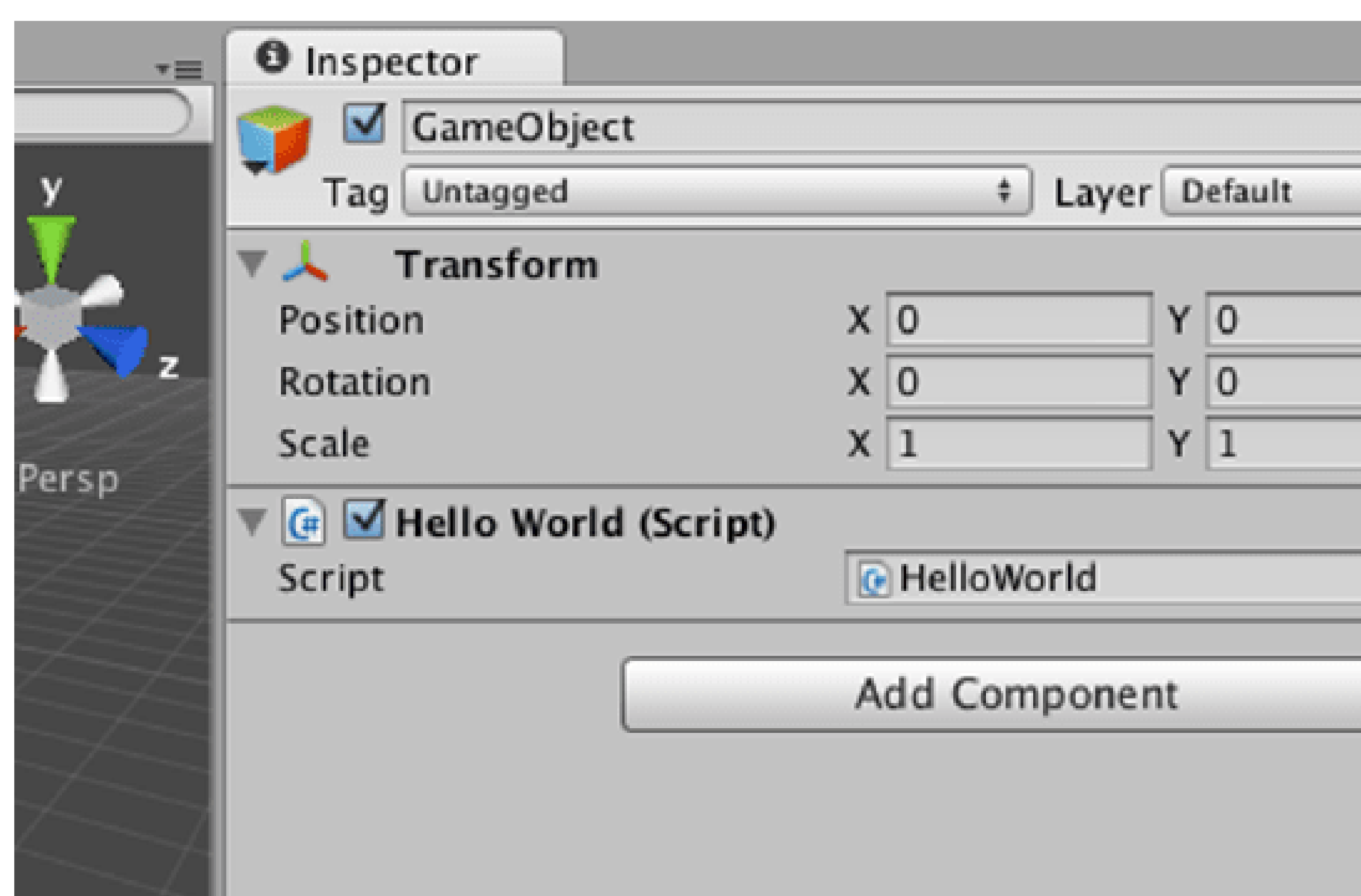


图 1-16　在 Inspector 中显示链接的脚本

代码清单 1.2　添加一个控制台消息

```
...
void Start() {
    Debug.Log("Hello World!");        ← 在此添加了日志命令
}
...
```

Debug.Log()命令将一个消息输出到 Unity 的 Console 视图中。与此同时，由于 Debug.Log 这一行代码出现在 Start()方法中，Start()方法会在对象激活时调用一次。换句话说，Start()方法会在单击编辑器中的 Play 时调用一次。一旦将日志命令添加到脚本中(请确认保存了脚本)，请单击 Unity 中的 Play 按钮并切换到 Console 视图，就会显示消息“Hello World!”。恭喜，第一个 Unity 脚本已经完成了！后续章节中的代码会更复杂，但这是重要的第一步。

“Hello World!”简明步骤

下面重申和总结一下前几页的步骤：

(1) 创建新项目。

(2) 创建新的 C#脚本。

(3) 创建空的 GameObject。

(4) 将脚本拖动到对象上。

(5) 给脚本添加日志命令。

(6) 单击 Play 按钮。

现在可以保存场景，这将创建一个带 Unity 图标的.unity 文件。场景文件是当前游戏中加载的任何东西的快照，因此可以在以后重新载入这个场景。保存这个场景没有什么价值，因为它太简单了(只是一个单一的空 GameObject)，但如果不保存这个场景，当退出 Unity 再回到这个项目时，就会发现它又变成空的。

脚本中的错误

为了查看 Unity 如何显示错误，故意在 HelloWorld 脚本中添加了一个拼写错误。例如，如果输入额外的圆括号，这个错误消息会出现在 Console 中，并带有红色的错误图标。如图 1-17 所示。

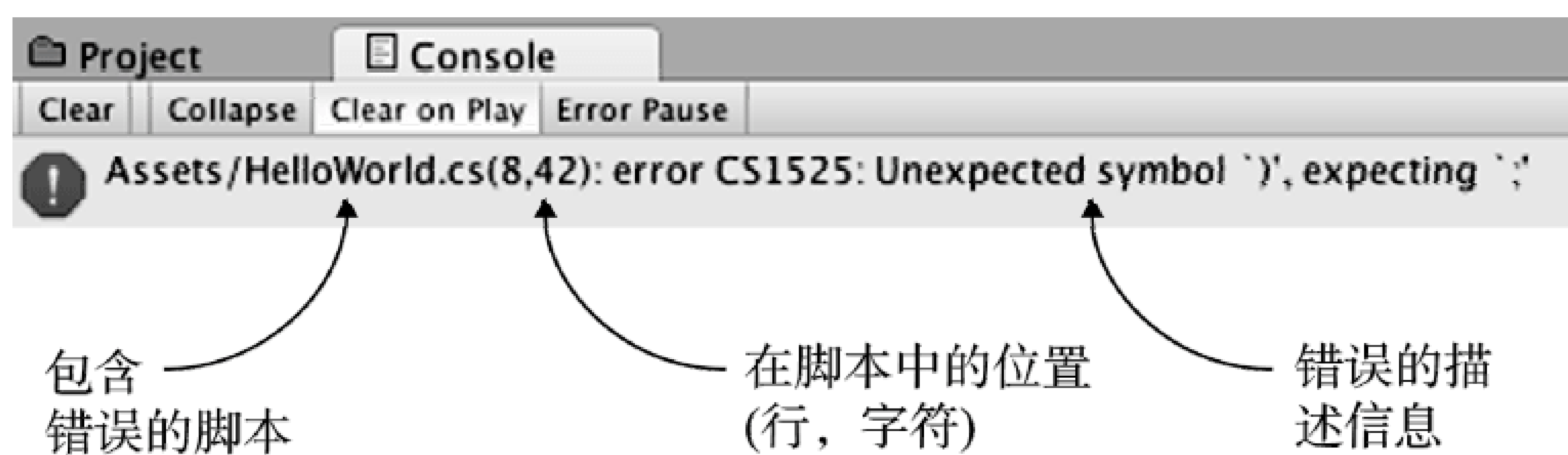


图 1-17　错误消息

1.4　小结

- Unity 是一个多平台的开发工具。
- Unity 的可视化编辑器包括可以协同工作的几部分。
- 脚本是作为组件附加到对象上的。
- 代码是使用 MonoDevelop 编写的内部脚本。

第 2 章

构建一个令人置身 3D 空间的演示游戏

本章涵盖：

- 了解 3D 坐标空间
- 在场景中放置一个玩家
- 编写移动对象的脚本
- 实现 FPS 控件

第 1 章以介绍传统的“Hello World！”程序结束。现在是时候介绍非传统的 Unity 项目了，这是一个带有交互和图形的项目。该项目将一些对象放到场景中，并编写代码，使玩家能在场景中走动。基本上，该项目就是一个没有怪物的 Doom(如图 2-1 所示)。Unity 中的可视化编辑器允许新用户立即构建 3D 原型，而不需要先编写大量模板代码(例如，初始化 3D 视图或建立渲染循环)。

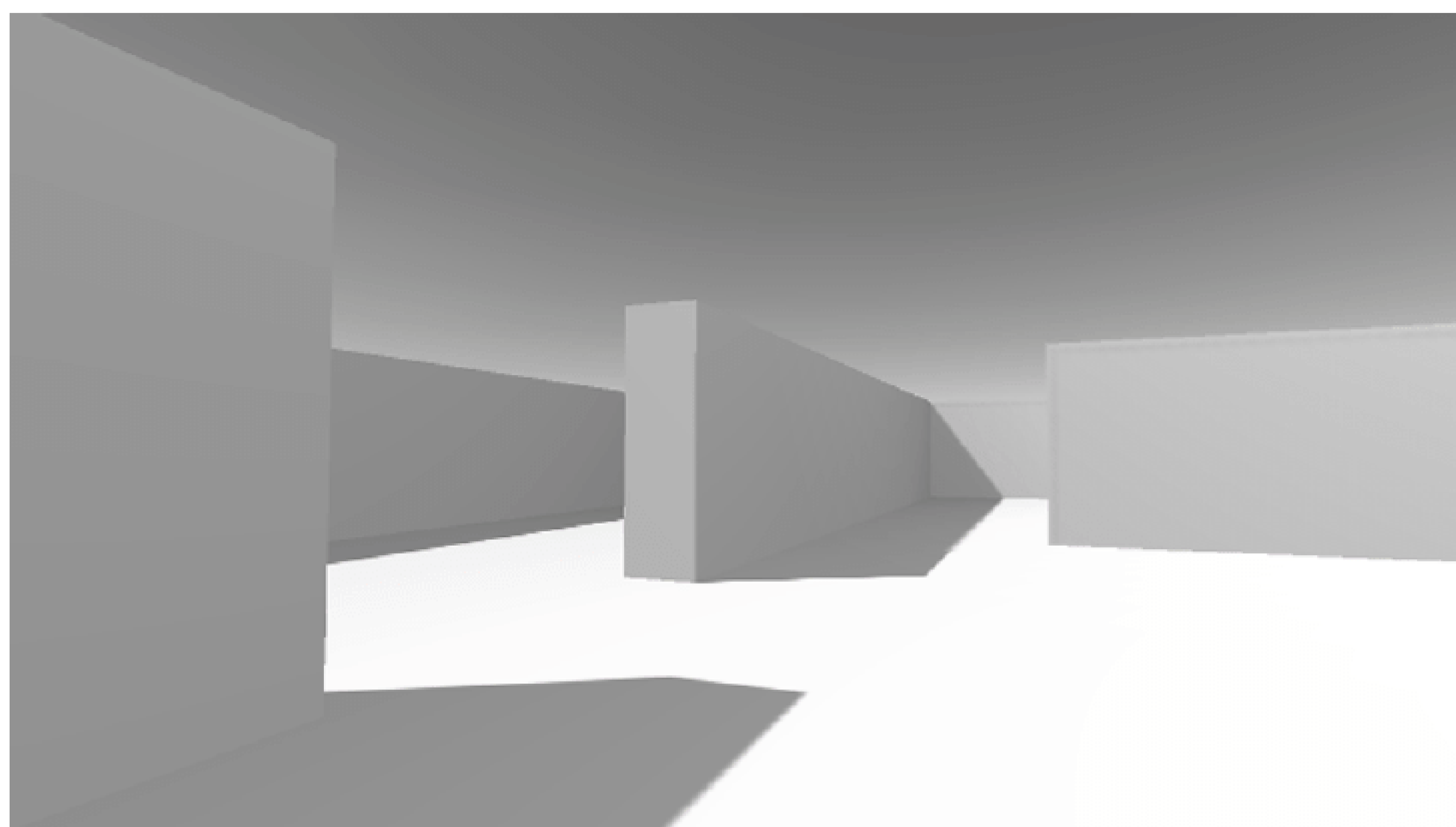


图 2-1　3D 演示游戏的截图(基本上就是没有怪物的 Doom)

在 Unity 中立刻开始构建场景很吸引人，尤其是如此简单(概念上的)的项目。但在开始时先停下来计划一下要做什么通常比较好，而这一步在现在尤其重要，因为读者还不熟悉这个流程。

注意　本章(和所有章节)的项目都可以从本书的网站上下载。在 Unity 中打开项目，然后打开 Scene 运行它。在学习过程中，建议自己键入所有的代码，下载的示例只用作参考。网站的地址是 www.manning.com/books/unity-in-action-second-edition。

2.1　在开始之前

Unity 使新手很容易上手，下面在构建整个场景前先复习一些知识点。尽管使用像 Unity 这样灵活的工具，也需要对工作的目标有一些了解。还需要掌握 3D 坐标的操作方式，否则当尝试在场景中定位一个对象时，很快就会迷失方向。

2.1.1　对项目做计划

在开始任何编程之前，通常需要问自己，“我现在在这里构建什么？”游戏设计本身就是一个宽泛的话题，有很多巨著专门阐述如何设计游戏。幸运的是，基于本例的目标，只需要对这个简单的演示游戏有个大概的了解，就能开发出基本的学习项目。这些入门项目不会有很复杂的设计，为让读者集中精力学习编程思想，可以(也应该)在掌握了游戏开发的原理之后，再了解高级的游戏设计理念。

第一个项目将构建一个基本的 FPS(First-Person Shooter，第一人称射击)场景。其中有一个玩家行走的房间，玩家将从其角色的视角看到游戏世界，玩家能通过鼠标和键盘控制角色。为了集中于核心机制——在 3D 空间中移动，先剥离整个游戏的所有有趣且复杂的内容。图 2-2 描述整个项目的路线图，该图基于作者大脑中构建的一个列表：

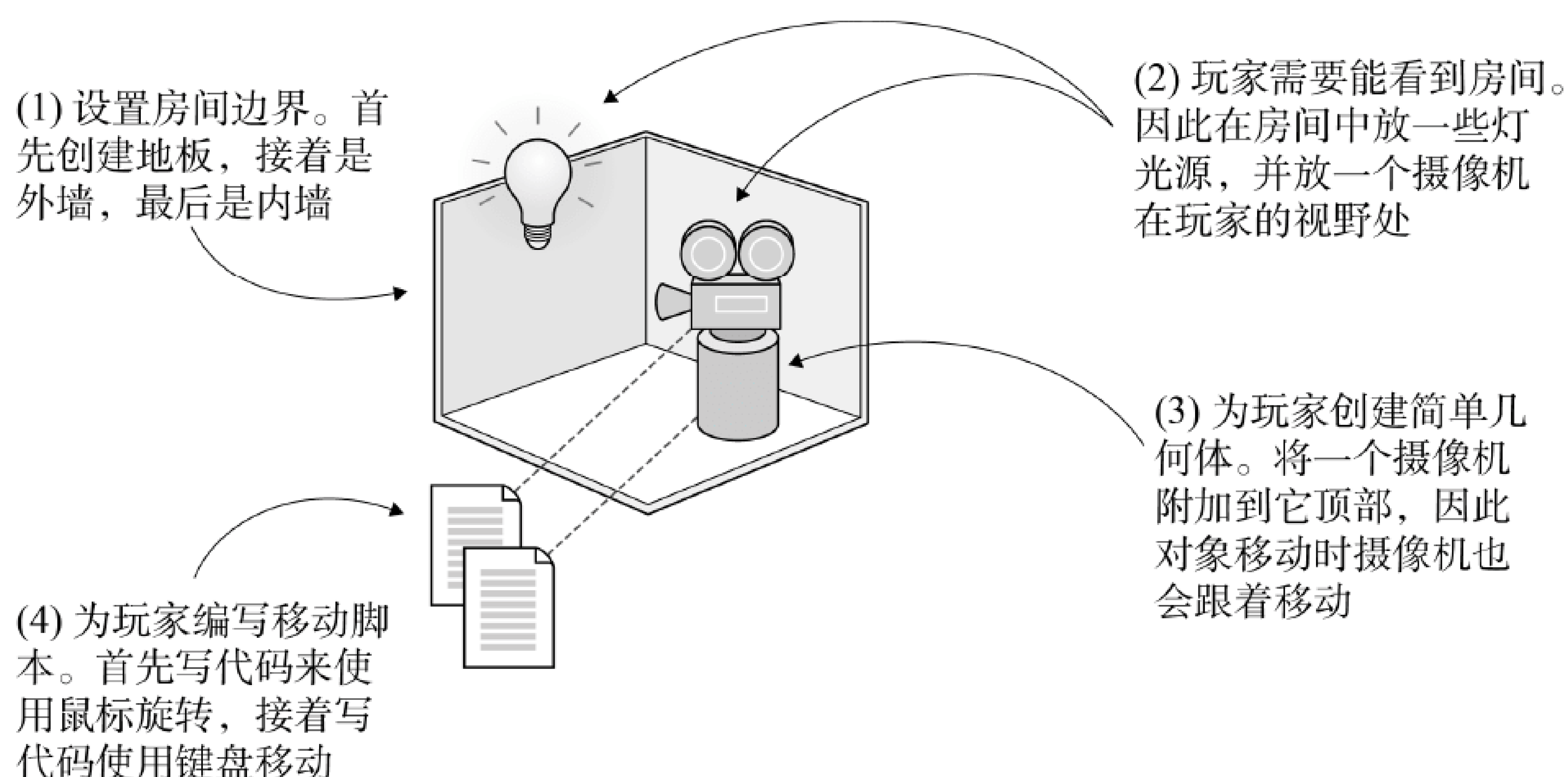


图 2-2　3D 演示游戏的路线图

不要被这个路线图吓跑！看起来好像本章涵盖了很多内容，但 Unity 会让它变得很简单。接下来关于移动脚本的部分篇幅会比较长，因为我们将会详细地讲解每一行，以使读者弄懂所有概念。这个项目是第一人称的演示游戏，美术资源的需求比较简单，因为我们看不到自己，所以使用顶部带摄像机的圆柱体来表示“玩家”！现在只要知道了 3D 坐标的工作原理，在可视化编辑器中放置任何东西都会很简单。

2.1.2　了解 3D 坐标空间

如果考虑当前正要实现的简单计划，就知道它包含三个方面：房间、视野和控制。这些事项要求理解 3D 计算机仿真是如何表达位置和移动的，而如果是刚开始从事 3D 图形工作，就可能还不知道这些内容。

其核心问题在于指示空间中的点的数字，这些数字通过坐标轴和空间关联起来。我们在数学课上，使用 X 轴和 Y 轴给纸上的点指定坐标(如图 2-3 所示)，即笛卡尔坐标系统。

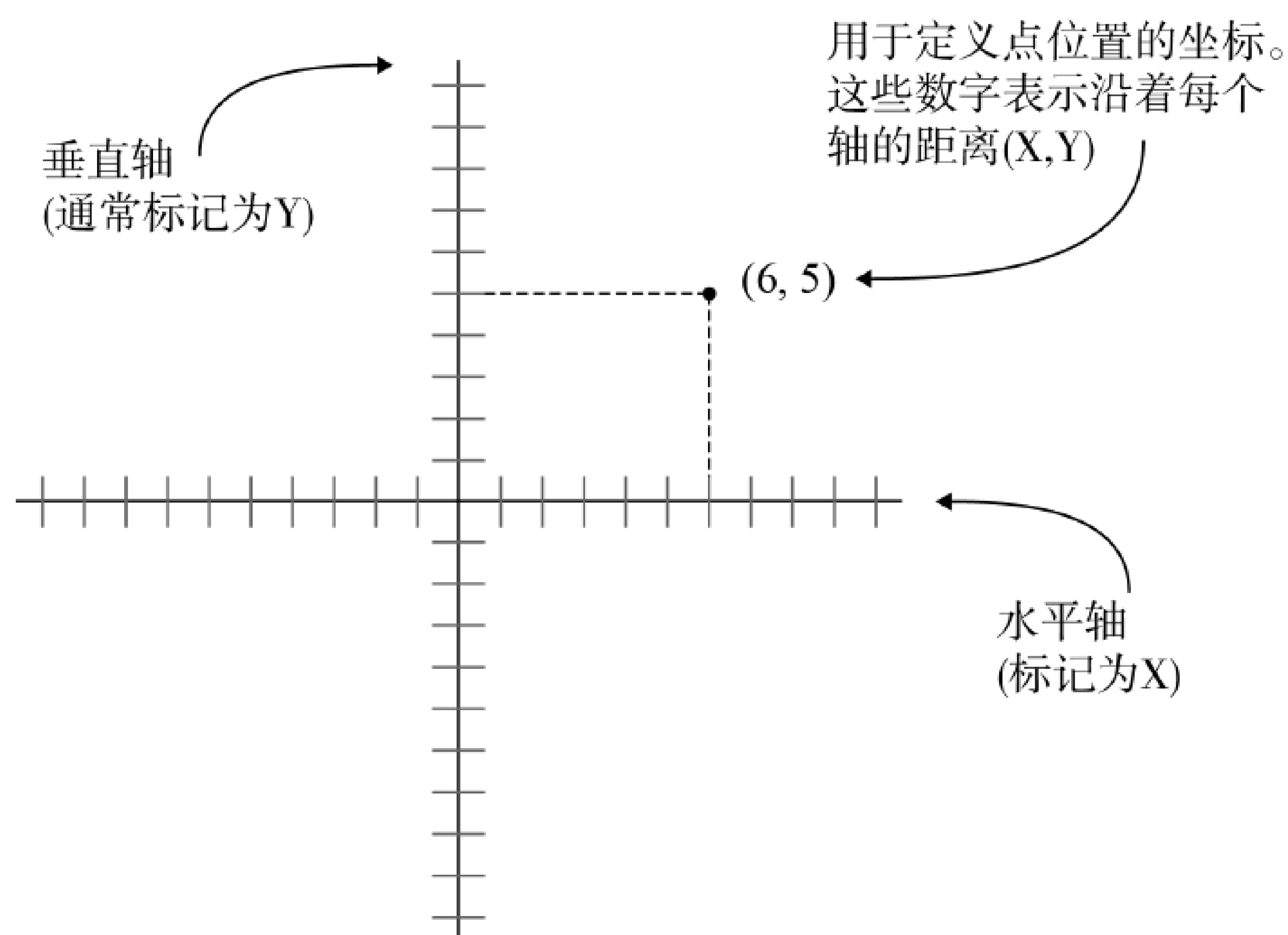


图 2-3　沿着 X 轴和 Y 轴的坐标定义了一个 2D 点

两个轴给出了 2D 坐标，所有点都在一个平面上。三个轴用于定义 3D 空间。X 轴沿着纸面的水平方向，Y 轴沿着纸面的垂直方向，我们现在想象有第三个轴，它垂直于纸面，并且指向纸外，同时垂直于 X 轴和 Y 轴。图 2-4 描绘了用于 3D 坐标空间的 X、Y、Z 轴。在场景中所有指定位置的东西都有 XYZ 坐标：玩家的位置，墙的放置等。

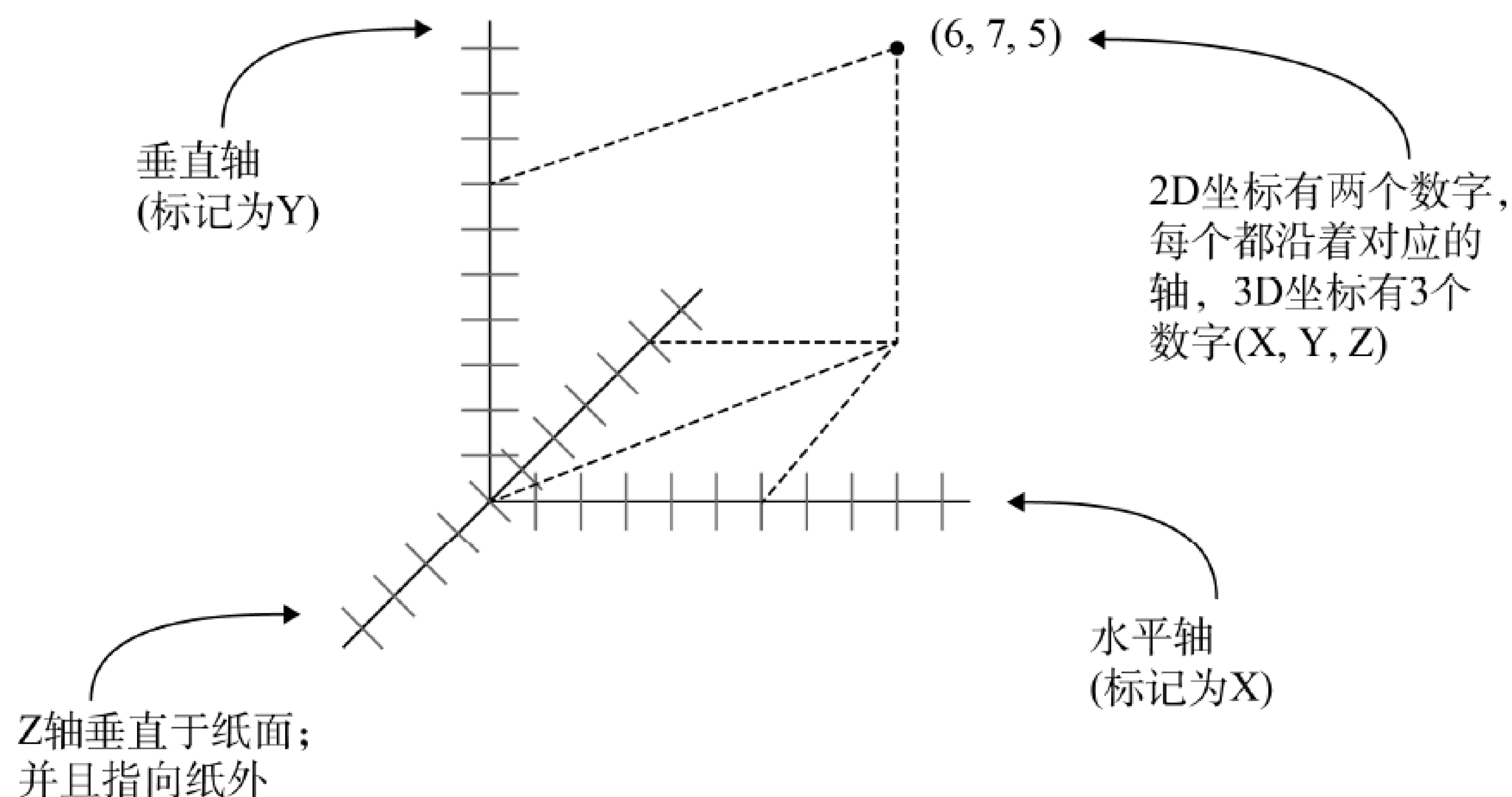


图 2-4 沿着 X，Y，Z 轴的坐标定义了一个 3D 点

在 Unity 的 Scene 视图中，显示了这三个轴，而在 Inspector 中，可以输入三个数字定位对象。不仅能写代码，使用三数字坐标定位对象，也能使用它们定义沿着每个轴移动的距离。

左手和右手坐标

每个轴的正方向和负方向是任意的，而不管轴的方向指向哪里，坐标都可以工作。只需要在给定的 3D 图形工具(动画工具、游戏开发工具等)中保持一致即可。

但大多数情况下，X 指向右，而 Y 指向上；不同工具之间的区别在于 Z 是指向纸里还是纸外。这两种方向分别称为“左手坐标”或“右手坐标”；如图 2-5 所示，如果拇指指向 X 轴，而食指指向 Y 轴，中指就指向 Z 轴。

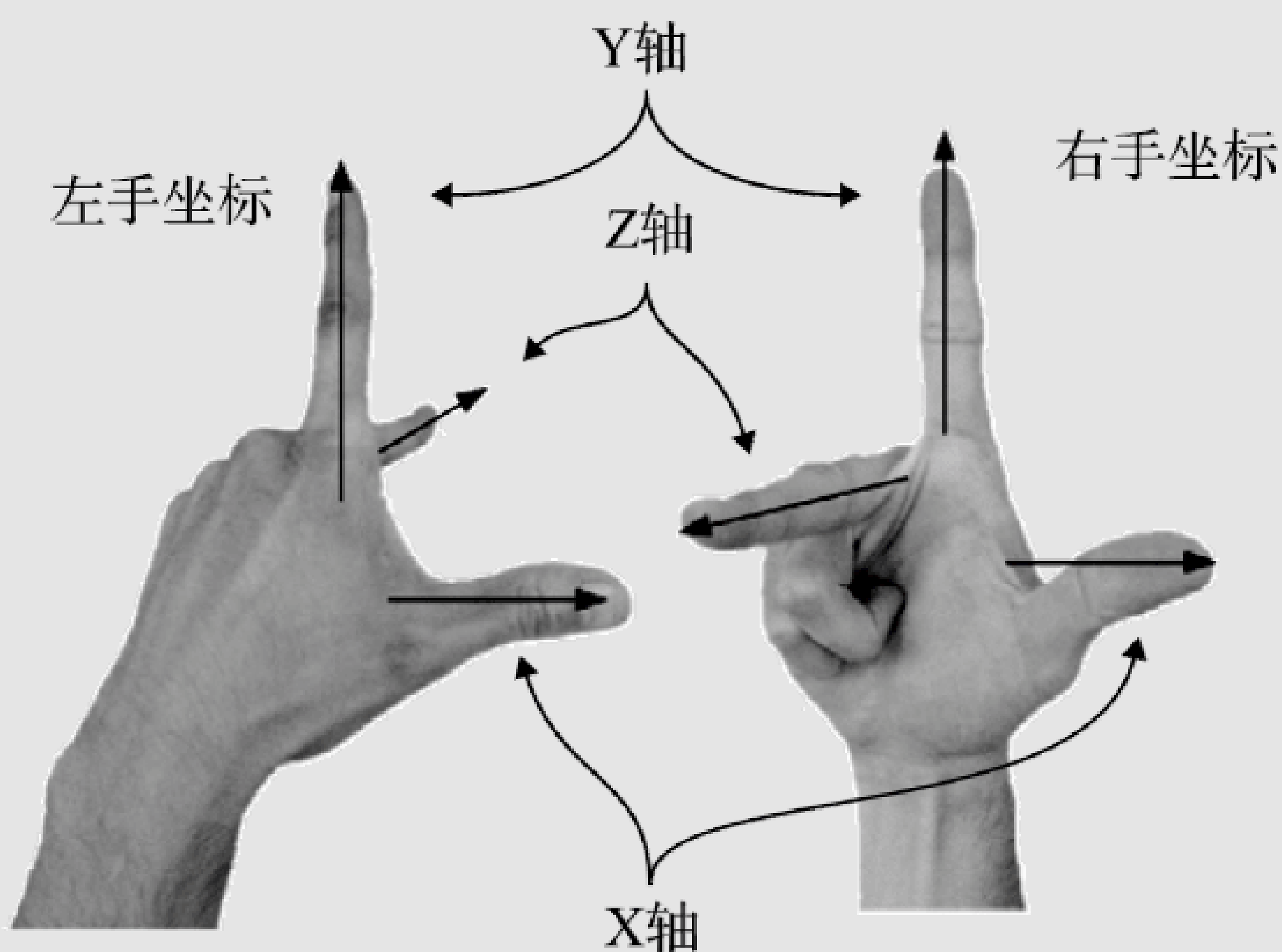


图 2-5 左手坐标和右手坐标的 Z 轴指向不同方向

Unity 和很多 3D 美术应用程序都使用左手坐标系统，很多其他的工具使用右手坐标系统(例如 OpenGL)。看到不同的坐标方向时，不要感到困惑。

现在已经有了项目的计划，知道如何使用坐标在 3D 空间中定位对象，就该构建场景了。

2.2　开始项目：在场景中放置对象

下面在场景中创建并放置对象。首先设置所有静态的布景——地板和墙。接着将沿着场景放置灯光，并定位摄像机。最后创建对象，即玩家，并在这个对象上附加脚本，使它在场景中移动。图 2-6 显示了一切就绪后编辑器中的场景。

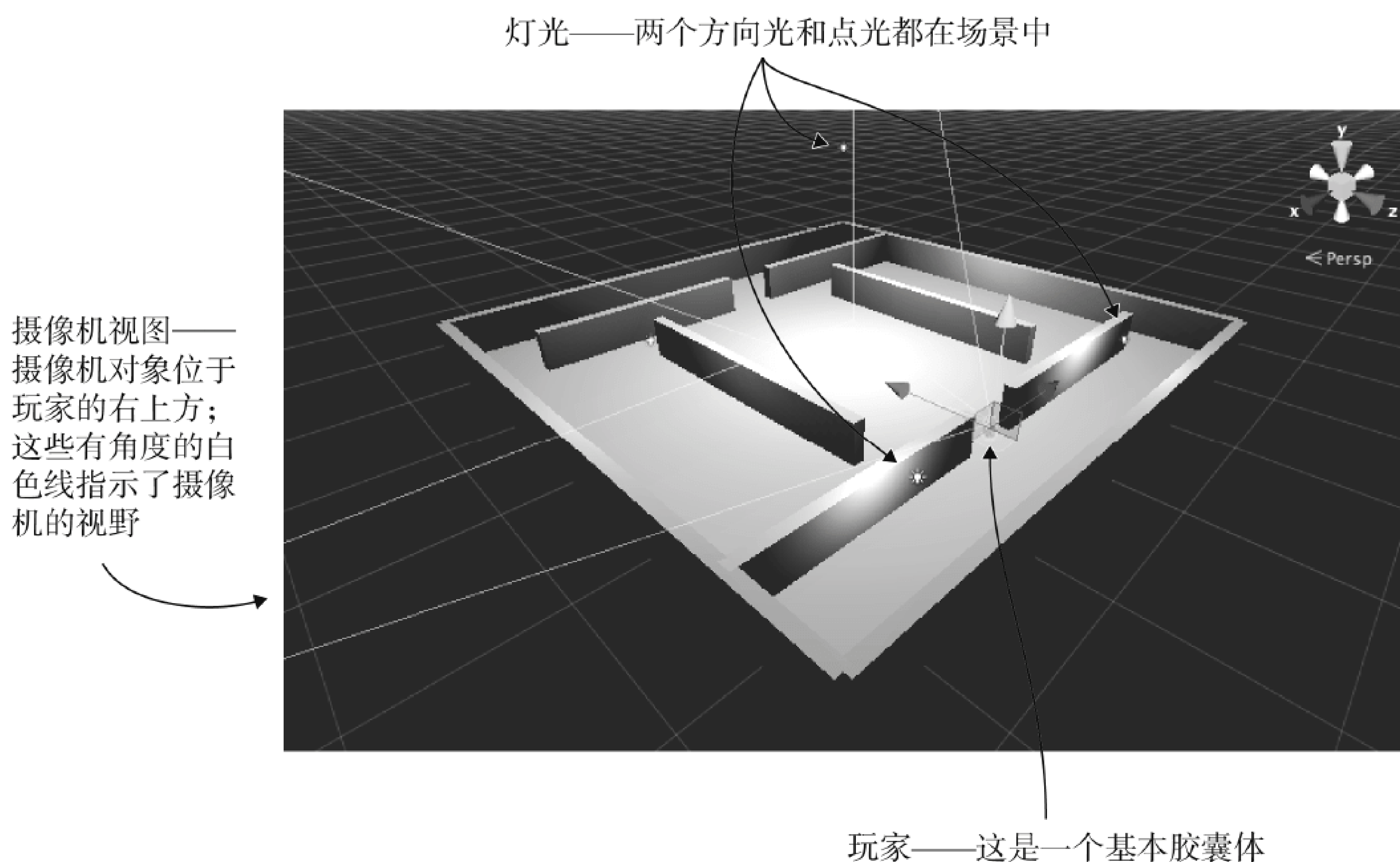


图 2-6　编辑器中的场景，包括地板、墙、灯光、摄像机和玩家

第 1 章说明了如何在 Unity 中创建一个新项目，现在就创建一个新项目。记住：选择 New(或者 File | New Project)，然后在弹出的窗口中为新项目命名。在创建新项目之后，立刻保存当前空的默认场景，因为项目不会保存任何初始化的场景文件。场景开始是空的，第一个要创建的对象是最显而易见的。

2.2.1　布景：地板、外墙和内墙

选择屏幕上方的 GameObject 菜单，将鼠标悬停到 3D Object 上，查看下拉菜单。然后选择 Cube，在场景中创建一个新的立方体对象(后面将使用其他形状，例如球体和胶囊体)。调整这个对象的位置、比例和名称来制作地板。图 2-7 展示了在 Inspector

中 floor 的值应该设置为多少(在拉伸它之前，它最初只是一个立方体)。

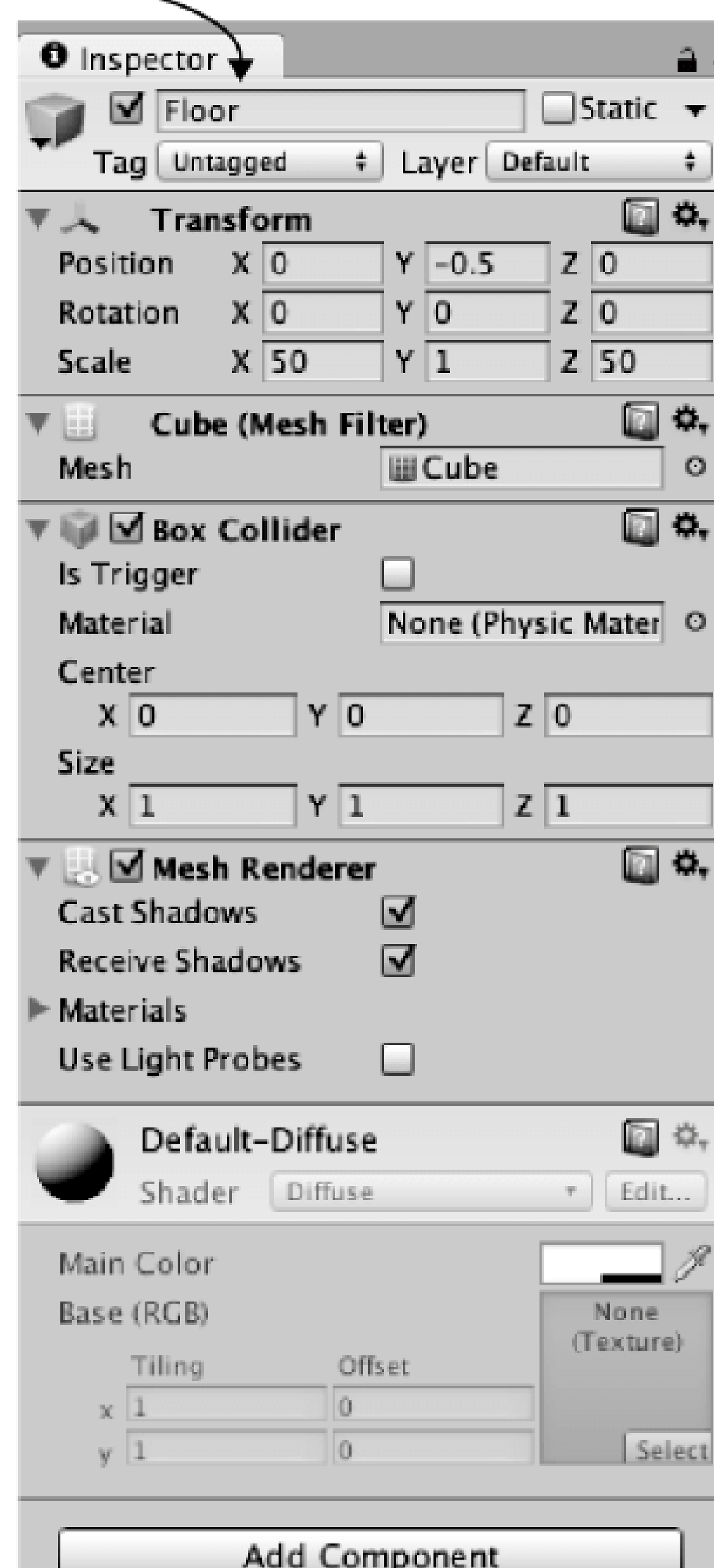


图 2-7　floor 的 Inspector 视图

注意　表示位置的数字可以是任意单位，只要单位在整个场景中保持一致即可。最常用的单位是米，这也是现在使用的单位，有时也使用步作为单位，甚至选择英寸作为单位!

重复刚才的步骤，给房间创建其他外墙。可以每次创建新的立方体，或者使用标准的快捷键来复制和粘贴已有的对象。移动、选择和缩放墙壁，形成地板的边界，尝试使用不同的数字(如 1，4，50 来缩放)或使用 1.2.2 节中介绍的变换工具(记住移动和旋转 3D 空间的数学术语是“变换”)。

提示　导航控件可以从不同的角度查看场景，或鸟瞰视图。如果在场景中迷失了，按 F 键可以将视野重置到当前选择的对象上。

墙壁的精确变换值取决于如何旋转和缩放立方体，也取决于对象在 Hierarchy 视图中彼此的关系。例如，在图 2-8 中，所有的墙都是一个空根对象的子对象。因此 Hierarchy 列表看起来比较有组织性。如果需要从一个例子中复制那些值，就下载示例项目，并参考项目中墙壁的变换值。

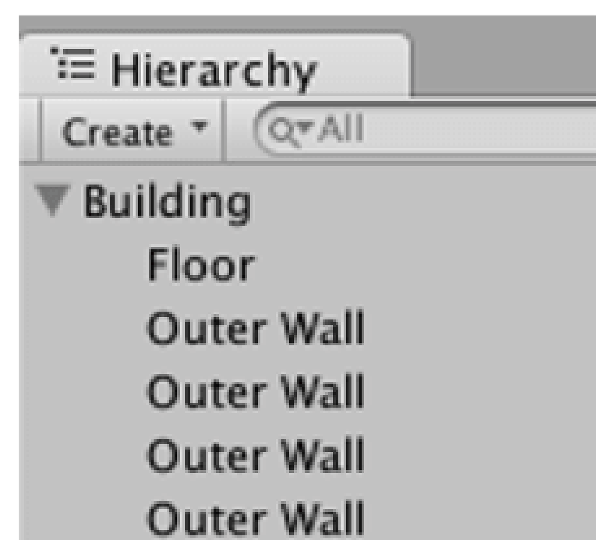


图 2-8　Hierarchy 视图显示由空对象组织的墙壁和地板

提示　在 Hierarchy 视图中拖动对象到另一个对象的上面可以建立连接。附着了其他对象的对象称为父对象；附着到其他对象上的对象称为子对象。移动(或者旋转和缩放)父对象时，子对象也会随之变换。

可以使用空的游戏对象以这种方式组织场景中的对象。将可见对象连接到一个根对象上，它们的 Hierarchy 列表就能够折叠。注意：在任何子对象连接到父对象上之前，都需要重置空的根对象的变换值[位置和旋转为(0，0，0)，缩放为(1，1，1)]，以避免以后出现任何定位错误。

什么是 GameObject?

所有场景对象都是类 GameObject 的实例，这类似于所有脚本组件都继承了类 MonoBehaviour。这个空对象的名称实际上也是 GameObject，无论该对象的名称是 Floor、Camera 还是 Player，该对象都是类 GameObject 的实例。

GameObject 实际上只是一些组件的容器。由于 GameObject 主要的用途是作为容器，因此可以把 MonoBehaviour 附加到它上面。对象在场景中具体是什么，取决于添加到 GameObject 上的是什么组件。Cube 对象有 Cube 组件，Sphere 对象有 Sphere 组件等。

一旦放置好外墙，就创建一些内墙用于导航。可以按照自己的意愿来放置这些墙；建议创建一些走廊和障碍物，这样一旦编写了移动代码，就可以绕着它们走。

现在场景中有一个房间，但里面没有任何灯光。下一步就解决这个问题。

2.2.2　灯光和摄像机

通常，在 3D 场景中使用一个平行光源点亮场景，然后再用一系列的点光源点亮场景。首先介绍平行光源。场景可能已经有一个默认的平行光源，但如果没有，则可以选择 GameObject | Light，然后选择 Directional Light，来创建平行光源。

光源的类型

可以创建一些类型的光源，这决定了它们如何并且往哪里投射光线。三种主要的光源是点光源、聚光源和平行光源。

点光源是一种从一点向所有方向射出光线的光源，就像真实世界中的灯泡。越靠近光源则越亮，因为光线在靠近光源的地方比较集中。

聚光源是一种从一点向一个有限的锥形发射光线的光源。这个项目没有使用聚光源，但这种灯通常用于关卡中的高亮部分。

平行光源是一种所有光线都平行、均匀的光源，场景中的所有对象都以相同的方式被照亮。这就像真实世界中的太阳。

平行光源的位置不会影响它发射的光，只影响光源面向的方向，所以在技术上，可以把平行光源放在场景中的任何位置。建议使它高过房间，这样它比较像太阳，而

且在操作场景中的其他对象时，不会遮挡它。旋转灯光，观察房间的效果，建议沿着 X 轴和 Y 轴稍稍旋转它，会获得较好的效果。在 Inspector 中可以看到 Intensity 设置(如图 2-9 所示)。顾名思义，这个设置控制灯光的亮度。如果这是唯一的光源，就必须更亮，但因为后面会增加一些点光源，所以这个平行光源可以暗一点，如设置 Intensity 为 0.6。

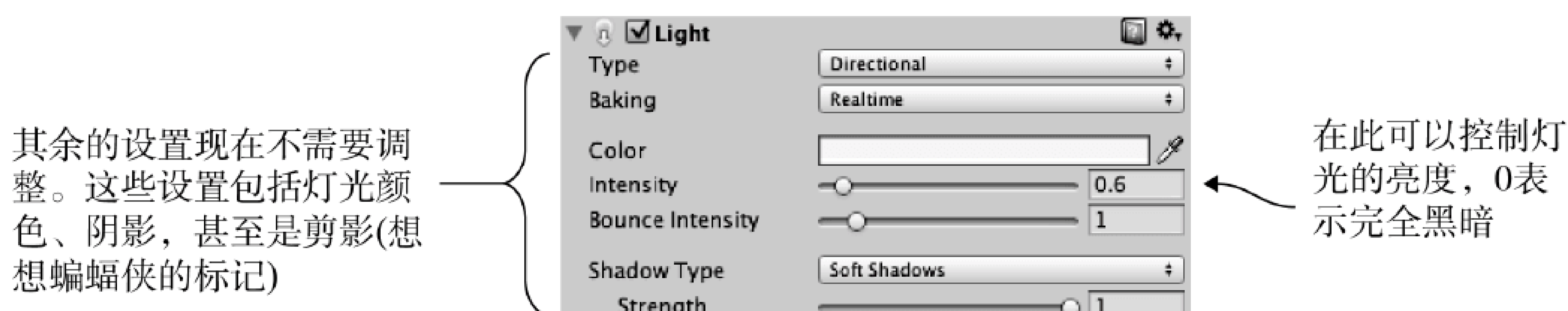


图 2-9　Inspector 中的平行光源设置

对于点光源，可以使用相同的菜单创建几个点光源，在房间的暗处放置它们，以确保所有墙都被照亮。不需要增加太多光源(当游戏有很多光源时，性能会降低)，但应该在每个角落都放置一个光源(建议把它们升到墙的顶部)，在场景的上方再增加一个光源(其 Y 坐标为 18)，让房间的灯光有一些变化。注意点光源的 Inspector(如图 2-10 所示)增加了对 Range(范围)的设置。这控制了光线能到达的距离；而平行光源发射的光线能到达整个场景，对象越靠近点光源就越亮，靠近地面的点光源的范围应该在 18 左右，放置在高处的点光源的范围应该在 40 左右，以照亮整个房间。

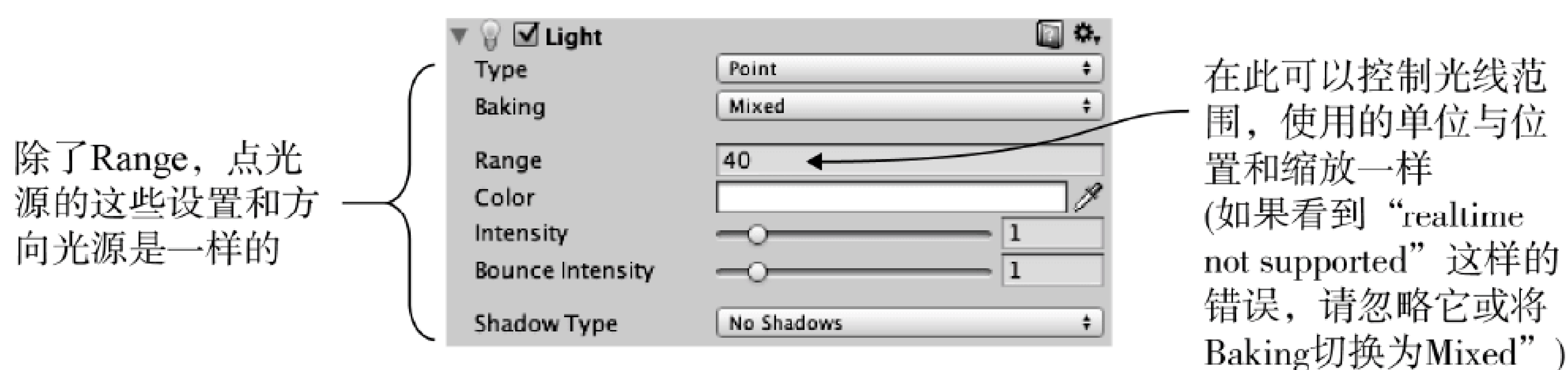


图 2-10　Inspector 中点光源的设置

为让玩家能看到场景，还需要另一种对象：摄像机(camera)，但“空”场景有一个主摄像机(main camera)，所以就使用这个主摄像机。如果需要创建新摄像机(例如在多人游戏中采用分屏视图)，Camera 同 Cube 和 Lights 一样是 GameObject 菜单的另一个选项。摄像机大致定位在玩家顶部，以便以玩家的视角观察视图。

2.2.3　玩家的碰撞器和视口

这个项目会用一个简单的几何体代表玩家。在 GameObject 菜单中(记住，把鼠标悬停在 3D Object 上，就会展开该菜单)单击 Capsule。Unity 就会创建一个圆角圆柱体。这个几何体代表玩家。设定这个对象的 Y 坐标为 1.1(对象高度的一半，增加一点高度可以避免和地板重叠)。沿着 X 轴和 Z 轴随意移动该对象，只要它在房间内，不碰到

任何墙壁即可。这个对象命名为 Player。

注意在 Inspector 中，这个对象被赋予了一个胶囊体碰撞器。这是胶囊体对象符合逻辑的默认选择，就像立方体对象默认有盒子碰撞器一样。但由于这个对象是玩家，因此需要的组件与大多数对象略有不同。单击该组件右上方的齿轮图标，移除胶囊体碰撞器，如图 2-11 所示；这会显示包含 Remove Component 选项的菜单。碰撞器是包围对象的绿色网格，所以删除胶囊体碰撞器时，绿色网格就会消失。

除了胶囊体碰撞器之外，还要给这个对象赋予一个角色控制器。在 Inspector 底部有一个 Add Component 按钮，单击该按钮，打开能添加的组件菜单。在菜单的 Physics 部分可以找到 Character Controller，选择该选项。顾名思义，这个组件将允许对象像一个角色那样动作。

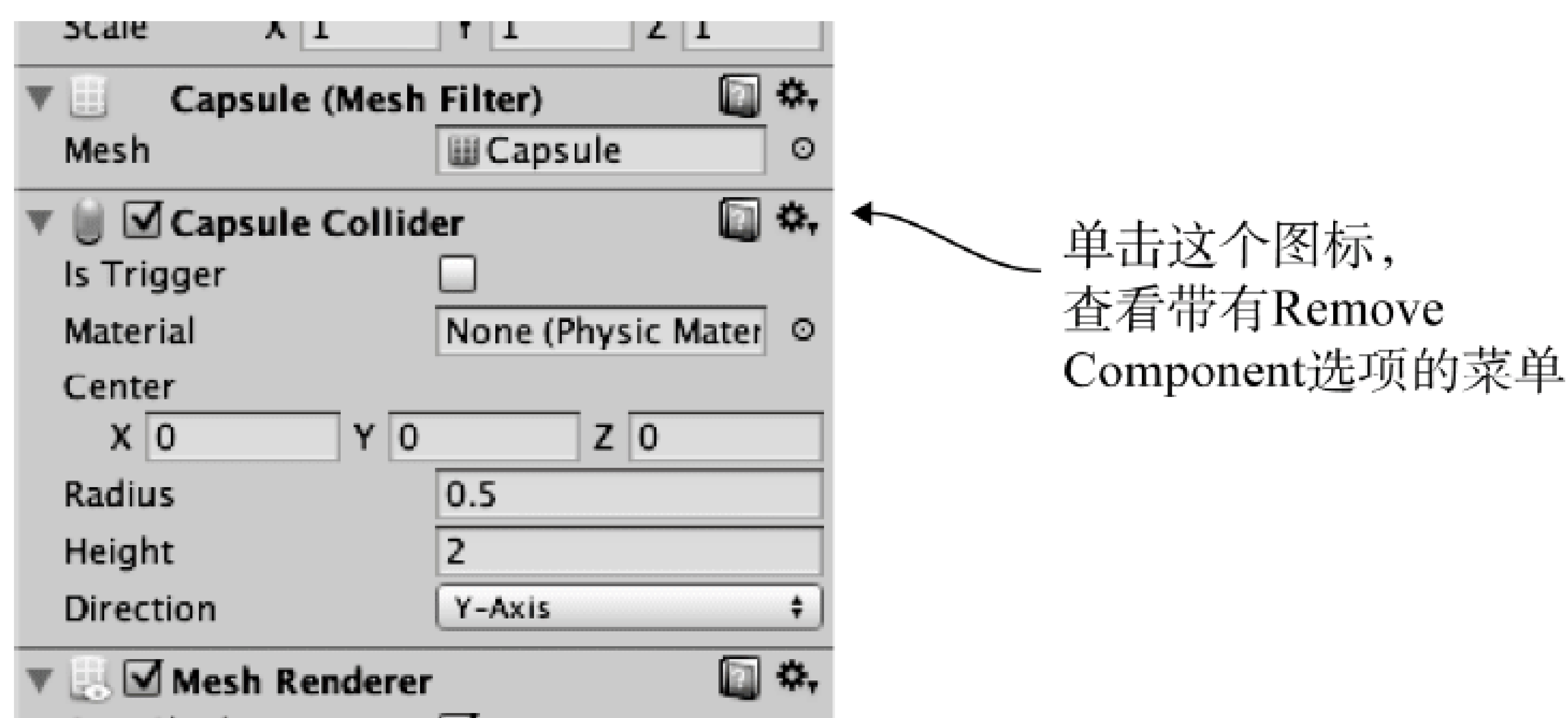


图 2-11　在 Inspector 中移除组件

设置玩家对象的最后一步是：附加摄像机。前面讨论地面和墙壁的章节中提到，可以在 Hierarchy 视图中将对象拖到另一个对象的上面。将摄像机对象拖动到玩家胶囊上，以将摄像机附加到玩家上。现在定位摄像机，让它看起来像是玩家的眼睛，建议位置是(0,0.5,0)。如有必要，摄像机的旋转重置为(0,0,0)(如果旋转过胶囊体，这个操作会去除摄像机的旋转设置)。

前面创建了这个场景需要的所有对象，剩下的任务就是编写代码，移动玩家对象。

2.3　移动对象：应用变换的脚本

为了让玩家在场景中移动，下面编写附加到玩家上的移动脚本。记住，组件是添加到对象上的模块化功能，脚本也是一类组件。最后这些脚本将响应键盘和鼠标的输入，不过首先是让玩家在场景中改变方向。这个简单的开头说明了如何在代码中应用变换。记住三个变换是 Translate、Rotate 和 Scale；旋转一个对象意味着改变它的旋转值。但这个任务除了“使对象旋转”之外，还有一些内容需要了解。

2.3.1　图示说明如何通过编程实现移动

实现对象的动画(例如让它旋转)归结于在每帧中都让它动一点，而这些帧会反复播放。由于应用变换是即时的，因此这明显和随着时间运动相反。通过一次次应用变换，使对象看起来像是在运动，就像一系列静止的图像在不停地翻页。图 2-12 显示了这是如何实现的。

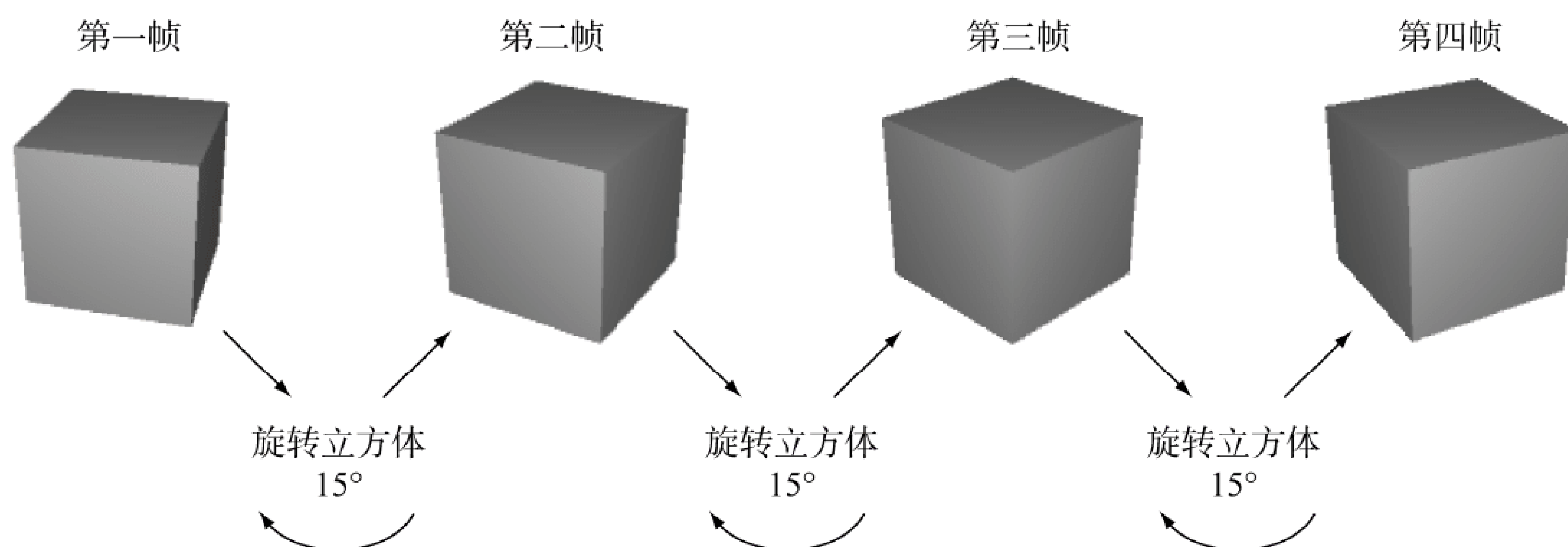


图 2-12　看起来在运动：静态图片间变换的周期性过程

回顾一下，脚本组件有 Update()方法，它会在每帧运行。为了旋转立方体，在 Update()中添加代码，使立方体每次旋转一个小的角度。所添加的代码会在每帧运行。听起来很简单，是吧？

2.3.2　编写代码实现图中演示的运动

现在将上述概念付诸实现。创建一个新的 C#脚本(记住在 Assets 菜单的子菜单 Create 中)，命名为 Spin，并编写代码清单 2.1(不要忘记在输入之后保存文件)。

代码清单 2.1　使对象旋转

```
using UnityEngine;
using System.Collections;
public class Spin : MonoBehaviour {
    public float speed = 3.0f;

    void Update() {
        transform.Rotate(0, speed, 0);
    }
}
```

声明一个公有变量，用于旋转速度

在此放置 Rotate 命令，以便它能在每帧运行

为将脚本组件添加到玩家对象上，从 Project 视图拖动脚本到 Hierarchy 视图的 Player 上。现在单击 Play 按钮，会看到视图在旋转；这就是让对象运动的代码！这段代码大多是新脚本的默认模板，只增加了两行新代码，下面解释这两行代码的作用。

首先，在类定义的顶部添加一个用于记录速度的变量(数字后的 f 告诉计算机把这个变量作为浮点值来处理，否则，C#会把小数看成双精度值)。旋转速度定义为变量，而不是常量，是因为 Unity 对脚本组件的公有变量做了一些便利的处理，如下面的“提示”中所述。

提示 公有变量显示在 Inspector 中，因此能在将组件添加给游戏对象之后再调整该组件的值。这称为“序列化”这个值，因为 Unity 会保存变量修改后的状态。

图 2-13 给出了脚本组件在 Inspector 中的显示情况。可以输入一个新数字，脚本将使用这个新数字，而不是代码中定义的默认值。这是一种便利的方式，可以调整组件在不同对象上的设置，在可视化编辑器中工作，而不是硬编码每个值。

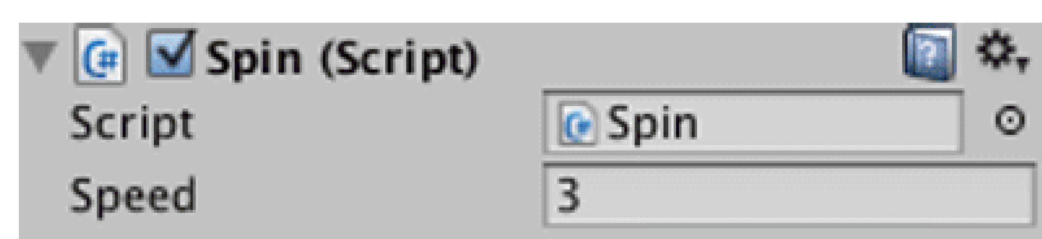


图 2-13　Inspector 显示脚本中声明的公有变量

代码清单 2.1 中的第二行是 Rotate()方法。它位于 Update()方法内，因此此命令会在每帧运行。Rotate()是 Transform 类的方法，所以它通过这个对象的变换组件，使用点符号来调用(在大多数面向对象语言中，this.transform 暗示着可以输入 transform)。这个变换操作是每帧旋转 speed 角度，得到平滑的旋转运动。但为什么 Rotate()的参数是(0, speed, 0)，而不是(speed, 0, 0)呢？

回想 3D 空间中有三个轴，X，Y 和 Z 轴。这些轴和位置、移动的关系是很直观的，这些轴也能用于描述旋转。航空学也用类似的方式描述旋转，所以 3D 图形学的编程人员通常借用航空学的一系列术语：航向偏角(pitch)、偏航(yaw)、侧滚(roll)。图 2-14 阐述了这些术语的意思：航向偏角是绕 X 轴旋转，偏航是绕 Y 轴旋转，侧滚是绕 Z 轴旋转。

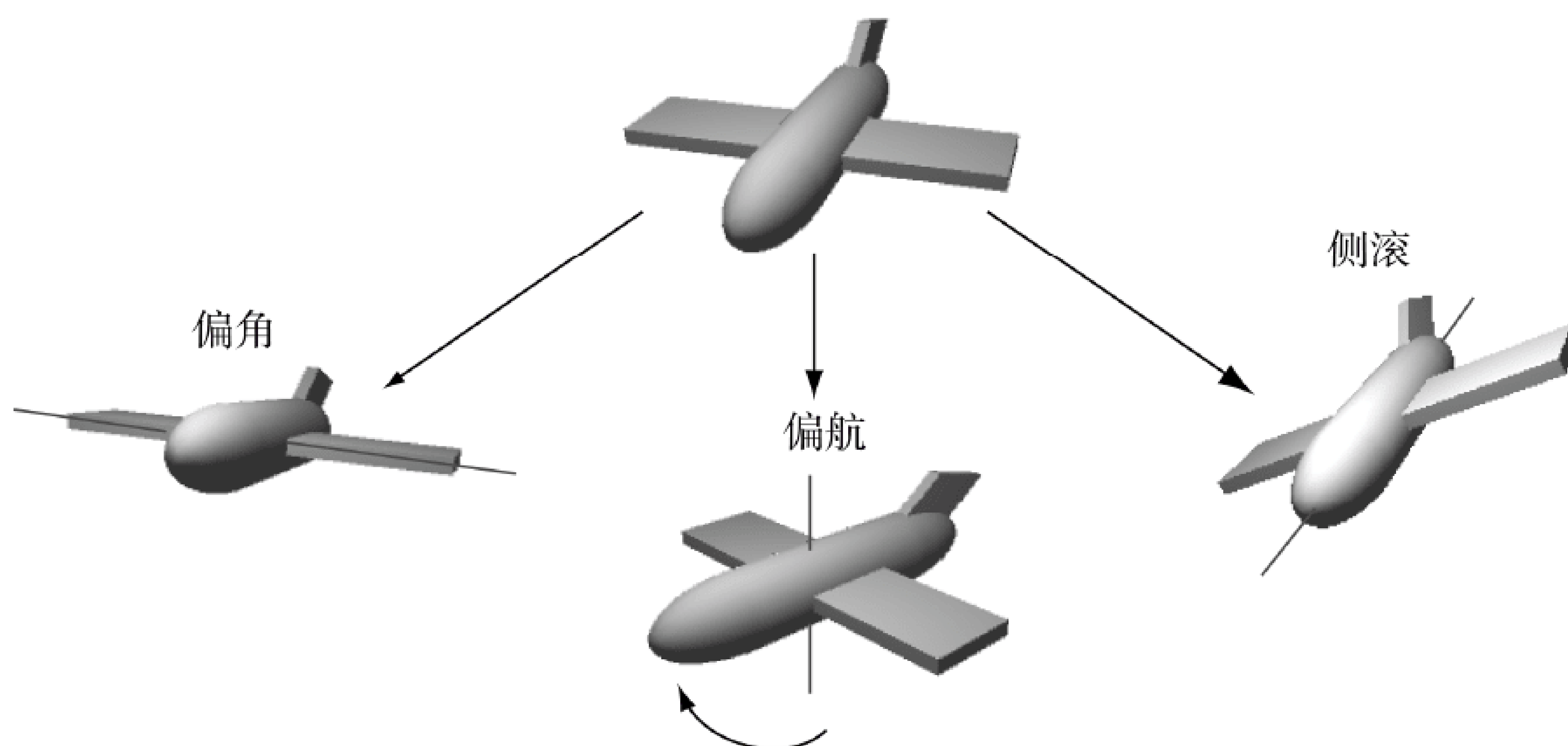


图 2-14　航向偏角、偏航和侧滚

假定绕 X，Y 和 Z 轴来描述旋转，这意味着 Rotate()的三个参数是 X、Y 和 Z 轴的旋转。因为我们只想让玩家绕着侧面旋转，而不是上下倾斜，所以只要给出 Y 轴的旋转值，X 和 Z 的旋转为 0 即可。如果参数改为(speed, 0, 0)，猜猜会发生什么情况？

现在就试着运行一下！

关于旋转和 3D 坐标轴还有一个微妙之处，体现在 Rotate()的第四个可选参数上。

2.3.3　本地和全局坐标空间

默认情况下，Rotate()方法基于本地坐标来操作。可以使用的另一类坐标是全局坐标。为可选的第四个参数输入 Space.Self 或 Space.World，可以告诉 Rotate()方法使用本地或者全局坐标。例如：

```
Rotate(0, speed, 0, Space.World)
```

参考本章前面对 3D 坐标空间的解释，考虑这些问题：(0, 0, 0)在哪里？X 轴指向哪里？坐标系统自己能移动吗？

事实证明，每个对象都有自己的原点，都有三个轴向，而且这个坐标系统会跟着对象一起移动。这样的坐标称为本地坐标。3D 场景也有自己的原点和自己的三个轴向，但这个坐标系统从不会移动。这样的坐标称为全局坐标。因此，为 Rotate()方法指定本地或全局坐标时，是在告诉该方法应绕哪个坐标轴的 X，Y，Z 轴旋转(如图 2-15 所示)。

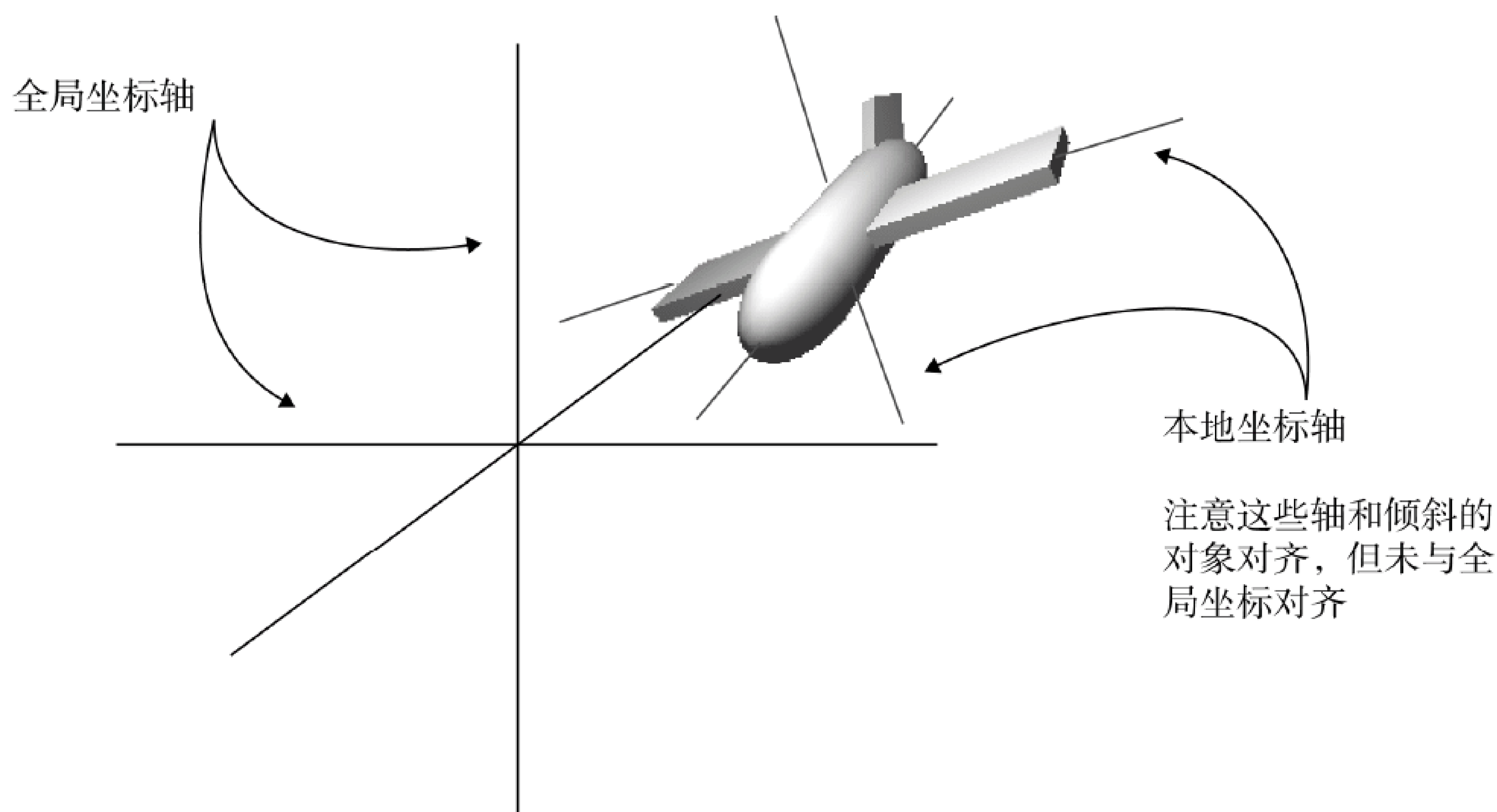


图 2-15　本地和全局坐标轴

如果刚接触 3D 图形，对这些概念多少会有点模糊。图 2-15 中描绘了两种不同的轴(注意飞机的“左边”和世界坐标的“左边”是不同的)，但通过一个例子来了解本地和全局坐标是最简单的方式。

首先，选择玩家对象，然后使他稍微倾斜(比如绕 X 轴旋转 30°)。这将把本地坐标抛离地面，因此本地和全局旋转看起来是不同的。现在尝试运行 Spin 脚本，分别给 Rotate()方法加上和不加上 Space.World 参数。如果很难观察发生了什么，请尝试从玩家对象上移除 Spin 组件，而是旋转一个放在玩家前面的倾斜立方体。命令设置为本地

或全局坐标时，对象将围绕不同的轴旋转。

2.4　用于观察周围的组件脚本：MouseLook

下面通过旋转来响应鼠标的输入(即旋转的对象是这个脚本附加到的对象，在这个例子中即玩家)。这要通过几步来实现，逐步给角色添加新的运动能力。首先玩家只能从一边旋转到一边，然后玩家能上下旋转。最后玩家能旋转到任意方向(水平旋转的同时也能垂直旋转)，这个行为称为鼠标观察(mouse-look)。

考虑到有三种不同类型的旋转行为(水平、垂直、水平且垂直)，首先将编写支持这三种旋转行为的框架。创建新的 C#脚本，命名为 MouseLook，并编写代码清单 2.2 中的代码。

代码清单 2.2　为 Rotation 设置使用枚举的 MouseLook 框架

```
using UnityEngine;
using System.Collections;
public class MouseLook : MonoBehaviour {
  public enum RotationAxes {
    MouseXAndY = 0,
    MouseX = 1,
    MouseY = 2
  }
  public RotationAxes axes = RotationAxes.MouseXAndY;

  void Update() {
    if (axes == RotationAxes.MouseX) {
      // horizontal rotation here
    }
    else if (axes == RotationAxes.MouseY) {
      // vertical rotation here
    }
    else {
      // both horizontal and vertical rotation here
    }
  }
}
```

定义枚举数据结构，将名称和设置关联起来

声明一个公有变量，以便在 Unity 编辑器中设置它

此处仅放置水平旋转的代码

此处仅放置垂直旋转的代码

此处放置水平且垂直旋转的代码

注意，使用枚举为 MouseLook 脚本选择水平或垂直旋转。定义枚举数据结构允许使用名称设置值，而不是输入数字并且尝试记住每个数字的意义(水平旋转是 0 还是 1？)。如果接着声明一个该枚举类型的公有变量，它在 Inspector 中将显示为下拉菜单(如图 2-16 所示)，这有利于选择设置。

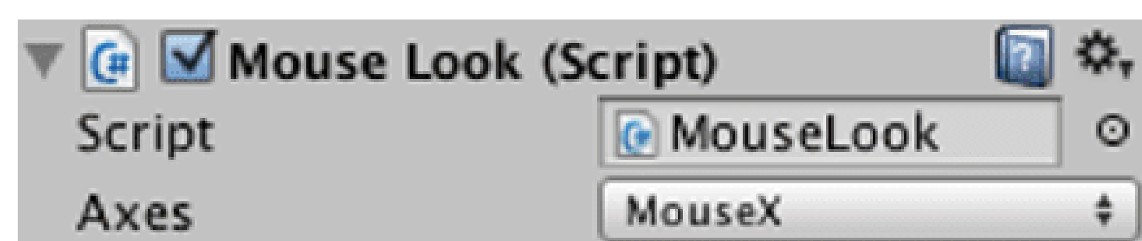


图 2-16　Inspector 将公有的枚举变量显示为下拉菜单

移除 Spin 组件(和之前移除胶囊体碰撞器的方法一样)，将这个新的脚本添加到

player 对象上。使用 Inspector 中的 Axes 下拉菜单切换旋转的方向。有了 horizontal/vertical 旋转设置后，就能为条件语句的每个分支填充代码。

2.4.1　跟踪鼠标移动的水平旋转

第一个且最简单的分支是水平旋转。先使用与代码清单 2.1 相同的旋转命令让对象旋转。不要忘记为旋转速度声明一个公有变量；在 axes 之后、Update()之前声明新变量，并把该变量命名为 sensitivityHor，因为一旦涉及多个旋转，speed 这个词就太普通了。这次把这个变量的值增加到 9，因为一旦代码开始缩放对象(稍后讨论)，这个值就需要更大。调整后的代码如代码清单 2.3 所示。

将 MouseLook 组件的 Axes 菜单设置为水平旋转，并运行脚本；视图将如之前一样旋转。下一步是让旋转响应鼠标的移动，所以需要介绍一个新方法：Input.GetAxis()。Input 类有一系列方法用于处理输入设备(例如鼠标)，而方法 GetAxis()返回和鼠标运动相关的数字(是正数还是负数，取决于移动的方向)。GetAxis()需要轴的名称作为参数，而水平轴称为 Mouse X。

代码清单 2.3　水平旋转，尚不能响应鼠标

```
...
public RotationAxes axes = RotationAxes.MouseXAndY;
public float sensitivityHor = 9.0f;

void Update() {
  if (axes == RotationAxes.MouseX) {
    transform.Rotate(0, sensitivityHor, 0);
  }
...
```

斜体代码已经在脚本中；在此显示只是为了参考

为旋转的速度声明一个变量

在此放置旋转命令，因此它能在每帧运行

如果将旋转速度乘以轴向的值，旋转将响应鼠标的移动。速度将根据鼠标的移动增加、缩小到 0 甚至是反向。Rotate 命令现在如代码清单 2.4 所示。

代码清单 2.4　为响应鼠标而调整的 Rotate 命令

```
...
transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
...
```

注意使用 GetAxis()获取鼠标的输入

单击 Play 按钮并四处移动鼠标。随着把鼠标从一边移向另一边，视图也将会从一边旋转到另一边。这样很酷！下一步介绍垂直旋转，而不是水平旋转。

2.4.2　有限制的垂直旋转

前面将 Rotate()方法用于水平旋转，但垂直旋转将使用不同的方法。尽管 Rotate()方法很便于应用变换，但不太灵活。它仅能用于没有限制地增加旋转值，这很适合水平旋转，但垂直旋转需要限制视野上下倾斜的程度。代码清单 2.5 展示了 MouseLook 中垂直旋转的代码，紧随其后的是代码的详细解释。

代码清单 2.5　MouseLook 的垂直旋转

```
...
public float sensitivityHor = 9.0f;
public float sensivityVert = 9.0f;      ← 为垂直旋转声明变量

public float minimumVert = -45.0f;
public float maximumVert = 45.0f;

private float _rotationX = 0;     ← 为垂直角度声明一个私有变量

void Update() {
  if(axes == RotationAxes.MouseX){
    transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
  }
  else if (axes == RotationAxes.MouseY) {
    _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;     ← 基于鼠标增加垂直角度
    _rotationX = Mathf.Clamp(_rotationX,minimumVert, maximumVert);     ← 将垂直角度限制在最小值和最大值之间

    float rotationY = transform.localEulerAngles.y;     ← 保持 Y 的角度不变(也就是水平没有旋转)

    transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);     ← 使用存储的旋转值创建新的向量
  }
}
```

把 MouseLook 组件的 Axes 菜单设置为垂直旋转，并运行新脚本。现在视图不会往两侧旋转，但当上下移动鼠标时它会上下倾斜。在上下限的位置停止倾斜。

这段代码中有几个新概念需要解释一下。首先，这次没有使用 Rotate()，所以需要一个变量(这里为_rotationX，因为垂直旋转绕 X 轴)来保存旋转的角度。Rotate()方法递增当前的旋转角度，而这段代码直接设置旋转的角度。换句话说，它的区别是“给角度增加 5”和“设置角度为 30”。旋转角度依然需要递增，这就是为什么代码有-=操作符：从旋转角度中减去一个值，而不是把旋转角度设置为那个值。若不使用 Rotate()，还可以以不同的方式处理旋转角度，而不仅仅是递增它。旋转值可以乘以 Input.GetAxis()，像水平旋转的代码一样，只是现在需要的是 Mouse Y，因为它是鼠标的垂直轴。

第二行代码进一步处理了旋转角度。Mathf.Clamp()用于将旋转角度保持在最小值和最大值之间。这些极值是之前代码声明的公有变量，它们确保视图只能上下倾斜 45°。Clamp()方法不只是用于旋转，它通常用于确保一个数值变量在限制的范围内。

要查看 Clamp()方法的作用，可以尝试注释掉 Clamp()那一行；现在倾斜不会在上下限处停止，甚至可以旋转到上下颠倒！显然，视图上下颠倒是不符合要求的，因此要对垂直旋转进行限制。

由于 transform 的 angles 属性是 Vector3，因此需要把旋转角度值传给构造函数，创建一个新的 Vector3。Rotate()方法会自动处理这一步，递增旋转角度并创建一个新向量。

定义　向量把多个数字存储为一个单元。例如，Vector3 有 3 个数字(称为 x, y, z)。

警告　需要创建一个新的 Vector3，而不是修改 transform 中已有的向量值，因为 transform 的那些值是只读的。这是一个常犯的错误。

欧拉角(Euler angle)和四元数(Quaternion)

为什么将属性命名为 localEulerAngles 而不是 localRotation？首先，需要了解四元数的概念。

四元数是描述旋转的另一个数学结构。它和欧拉角不同，欧拉角是之前采用的 X、Y、Z 轴的方法的名称。还记得航向偏角、偏航和侧滚的讨论吗？这种描述旋转的方法便是欧拉角。四元数和欧拉角不同。很难解释四元数是什么，因为它是高等数学中的一个晦涩的概念，涉及通过四维表示运动。详细解释请参阅以下网址：

www.flipcode.com/documents/matrfaq.html-Q47

对于为什么四元数用于表示旋转有个比较简单的解释：使用四元数在旋转值之间插值(就是通过一些中间值来慢慢从一个值变为另一个值)看起来更平滑、自然。

回到最初的问题，这是因为 localRotation 是一个四元数，而不是欧拉角。而 Unity 也提供欧拉角属性，让旋转的处理更容易理解；因此使用 localEularAngles 命名旋转属性。欧拉角属性和四元数之间可以来回自动转换。Unity 在后台自动处理数学难题，不必自己去处理。

MouseLook 还有一个旋转设置需要编写代码：同时水平和垂直旋转。

2.4.3　同时水平旋转和垂直旋转

最后一块代码也不使用 Rotate()，其原因和前面相同：垂直旋转角度在递增之后要限制在某个范围内。这意味着水平旋转现在也需要直接计算。记住，Rotate()会自动递增旋转角度(查看代码清单 2.6)。

代码清单 2.6 MouseLook 中的水平且垂直旋转

```
...
else {
  _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
  _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

  float delta = Input.GetAxis("Mouse X") * sensitivityHor;
  float rotationY = transform.localEulerAngles.y + delta;

  transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
}
...
```

delta 是旋转的变化量

使用delta 递增旋转角度

处理_rotationX 的前几行代码和代码清单 2.5 完全一样。只是要记住绕对象的 X 轴的旋转是垂直旋转。因为水平旋转不再通过 Rotate()方法处理，而是由 delta 和 rotationY 代码行完成。delta 是一个通用的数学术语，表示“变化量”，因此 delta 计算的正是旋转角度应该改变的量。接着把该变化量加到当前的旋转角度上，得到最新的旋转角度。

最后使用绕水平轴和垂直轴旋转的角度值创建一个新的向量，接着将它赋值给变换组件的角度属性。

禁止对玩家进行物理旋转

尽管这对这个项目还不需要，但在大多数现代 FPS 游戏中，有一个复杂的物理仿真会影响场景中的所有对象，导致对象被弹开和跌倒。这种碰撞的行为看起来不错，很适合大多数对象，但玩家的旋转需要由鼠标单独控制，不能受该物理仿真的影响。

因此，鼠标输入脚本通常在玩家的 Rigidbody 上设置 freezeRotation 属性。将下面的 Start()方法添加到 MouseLook 脚本中：

```
...
void Start(){
    Rigidbody body = GetComponent<Rigidbody>();
    if (body != null)
        body.freezeRotation = true
}
```

检查这个组件是否存在

Rigidbody(刚体)是对象能拥有的一个额外组件。物理仿真作用于 Rigidbody，并处理它们附加到的对象。

为防止忘记对脚本进行的各种修改和添加的内容，下面复习一下，代码清单 2.7 给出了完整的代码，也可以下载该示例项目。

代码清单 2.7 完成的 MouseLook 脚本

```
using UnityEngine;
using System.Collections;
```

```
public class MouseLook : MonoBehaviour {
  public enum RotationAxes {
    MouseXAndY = 0,
    MouseX = 1,
    MouseY = 2
  }
  public RotationAxes axes = RotationAxes.MouseXAndY;

  public float sensitivityHor = 9.0f;
  public float sensitivityVert = 9.0f;

  public float minimumVert = -45.0f;
  public float maximumVert = 45.0f;

  private float _rotationX = 0;

  void Start() {
    Rigidbody body = GetComponent<Rigidbody>();
    if (body != null)
        body.freezeRotation = true;
  }

  void Update() {
    if (axes == RotationAxes.MouseX) {
      transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
    }
    else if (axes == RotationAxes.MouseY) {
      _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
      _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

      float rotationY = transform.localEulerAngles.y;

      transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
    }
    else {
      _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
      _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

      float delta = Input.GetAxis("Mouse X") * sensitivityHor;
      float rotationY = transform.localEulerAngles.y + delta;

      transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
    }
  }
}
```

当设置 Axes 菜单并运行新代码时，可以在移动鼠标时观看周围的所有方向。但依然卡在一个地方，好像被固定在一个炮塔上。下一步是在场景中移动。

2.5 键盘输入组件：第一人称控件

响应鼠标输入来观察四周是第一人称控件中一个重要的部分，但仅完成了一半。

玩家也需要移动，来响应键盘输入。下面编写键盘控件组件来补充鼠标控件组件；新创建一个称为 FPSInput 的 C#脚本，并把它附加到玩家上(在 MouseLook 脚本旁边)。目前 MouseLook 组件暂时设置为只做水平旋转。

提示　这里讲解的键盘和鼠标控件被分离到单独的脚本中。可以不用这种方式组织代码，而将所有内容打包到一个 player controls 脚本中，但将功能切分到每个小组件中，组件系统(诸如 Unity 的组件系统)将最灵活、最高效。

前一节编写的代码只影响旋转，现在要改变对象的位置。把代码清单 2.1 输入 FPSInput 中，但把 Rotate()改为 Translate()。当单击 Play 按钮时，视图会上升而不是旋转。尝试修改参数的值，看看运动是如何改变的(特别是尝试交换第一个和第二个参数)；在做了上述实验后，可继续添加键盘输入的代码。如代码清单 2.8 所示。

代码清单 2.8　使用代码清单 2.1 的旋转代码，并做些许修改

```
using UnityEngine;
using System.Collections;

public class FPSInput : MonoBehaviour {
  public float speed = 6.0f;              // 不是必要的，但可能想要增加速度

  void Update(){
    transform.Translate(0, speed, 0);     // 将 Rotate()修改为 Translate()
  }
}
```

2.5.1　响应按下的键

根据按下的键移动的代码(如代码清单 2.9 所示)类似于根据鼠标旋转的代码。这里也以相似的方式使用 GetAxis()方法。代码清单 2.9 展示了如何使用 GetAxis 命令。

代码清单 2.9　响应按键而移动位置

```
...
void Update() {
  float deltaX = Input.GetAxis("Horizontal") * speed;   // Horizontal 和 Vertical 是键盘映射的间接名称
  float deltaZ = Input.GetAxis("Vertical") * speed;
  transform.Translate(deltaX, 0, deltaZ);
}
...
```

如前所述，GetAxis()的值乘以速度就确定移动量。以前请求的轴都是“Mouse something”，而现在传入 Horizontal 或 Vertical。这些名称是 Unity 中输入设置的抽象；在 Project Settings 下的 Edit 菜单中，有一个 Input 菜单，其中列出抽象输入名称和映射到那些名称的具体控件。左/右箭头按键和字母键 A/D 都映射到 Horizontal，而上下箭头按键和字母键 W/S 都映射到 Vertical。

注意，移动的值应用到 X 和 Z 坐标。在实验 Translate()方法时可能注意到，X 坐标从屏幕一边移动到另一边，Z 坐标从前面移动到后面。

输入下面新的移动代码，则按下箭头键或 WASD 字母键就可以四处移动，这是大多数 FPS 游戏的标准。移动脚本就要完成了，但还要做一些调整。

2.5.2　设置独立于计算机运行速度的移动速率

现在还不明显，因为代码只是在一台(自己的)电脑上运行，但如果在不同的机器上运行代码，代码运行的速度则会不同。这就是为什么一些计算机处理代码和图形的速度比其他计算机快的原因所在。现在玩家在不同的计算机上会以不同的速度移动，因为移动代码是根据计算机的速度决定的。这称为帧率依赖(frame rate dependent)，因为移动代码依赖于游戏的帧率。

假定在两台不同的计算机上运行这个示例，一台计算机的速率是 30 帧/秒，而另一台是 60 帧/秒。这意味着在第二台计算机上，Update()的调用频率是第一台的两倍，而每次都使用相同的速度值 6。在 30 帧/秒的机器上，移动速度会是 180 单位/秒，而在 60 帧/秒的机器上，移动速度则是 360 单位/秒。对于大多数游戏而言，这样的速度不同其实并不是好事。

解决方案是调整移动代码，使它独立于帧率。这意味着移动速度不依赖游戏的帧率。为此就不能在每帧率应用相同的速度值。而是根据计算机运行的快慢提高或降低速度值。为此应把速度值和另一个称为 deltaTime 的值相乘，如代码清单 2.10 所示。

代码清单 2.10　使用 deltaTime 使移动独立于帧率

```
...
void Update() {
  float deltaX = Input.GetAxis("Horizontal") * speed;
  float deltaZ = Input.GetAxis("Vertical") * speed;
  transform.Translate(deltaX * Time.deltaTime, 0, deltaZ * Time.deltaTime);
}
...
```

上面的修改很简单。Time 类有一些用于计算时间的属性和方法，而其中就包括属性 deltaTime。delta 意味着变化量，这说明 deltaTime 是时间的变化量。明确地说，deltaTime 是经过两帧之间的时间。在不同的帧率下，两帧之间的时间是不同的(例如，对于 30 帧/秒，deltaTime 是每秒的 1/30)，因此把速度值乘以 deltaTime，将提高或降低不同计算机上的速度值。

现在移动速度在所有的计算机上都是一样的，但是移动脚本还没有全部完成；在房间中四处移动时，还能穿过墙，因此需要调整代码，来阻止这种情况。

2.5.3　移动 CharacterController 以检测碰撞

直接修改对象的变换，不会应用碰撞检测，因此角色将穿过墙。为了应用碰撞检测，需要使用 CharacterController。这个组件会让对象移动起来更像是游戏中的角色，包括和墙壁碰撞。回想一下，设置玩家时，附加了一个 CharacterController，所以现在提供 FPSInput 中的移动代码来使用该组件(如代码清单 2.11 所示)。

代码清单 2.11　使用 CharacterController 替代 Transform

```
...
private CharacterController _charController;

void Start() {
  _charController = GetComponent<CharacterController> ();
}

void Update() {
  float deltaX = Input.GetAxis("Horizontal") * speed;
  float deltaZ = Input.GetAxis("Vertical") * speed;
  Vector3 movement = new Vector3 (deltaX, 0, deltaZ);
  movement = Vector3.ClampMagnitude (movement, speed);

  movement *= Time.deltaTime;
  movement = transform.TransformDirection (movement);
  _charController.Move (movement);
}
...
```

用于引用 CharacterController 的变量

使用附加到相同对象上的其他组件

使对角移动的速度和沿轴移动的速度一样

告知 CharacterController 通过 movement 向量移动

把 movement 向量从本地坐标变换为全局坐标

这段代码引入一些新概念。第一个要指出的概念是引用 CharacterController 的变量。这个变量创建一个到对象的本地引用(代码对象——不要和场景对象混淆)；多个脚本都能引用这个 CharacterController 实例。

变量开始是空的，因此在使用这个引用之前，需要将一个对象赋给它，让它指向对象。这就是 GetComponent()的作用，这个方法返回附加到相同 GameObject 上的其他组件。不是将参数传入圆括号中，而是使用 C#在尖括号<>中定义类型的语法。

一旦有了 CharacterController 的引用，就能调用控制器的 Move()方法。向 Move()传入一个向量，就像鼠标旋转代码使用一个向量作为旋转值一样。同时像限制旋转值一样，使用 Vector3.ClampMagnitude()把向量的大小限制为移动速度。在此使用 clamp，否则对角移动将比沿着轴移动的速度大(想象直角三角形的直角边和斜边)。

但此处的移动向量还有一个棘手的地方，它与使用本地坐标还是全局坐标有关，如之前讨论的旋转所述。下面创建一个向量用于移动，其中的一个值表示左移。这里是指玩家的左边，然而，它的方向可能和全局坐标的左边完全不同。即我们讨论的左边是本地空间中的，而不是全局空间中的。需要给 Move()方法传入一个在全局空间中

定义的移动向量，因此需要把本地空间向量转为全局空间的向量。进行这个转换是一个很复杂的数学过程，但幸好 Unity 会自动完成这个数学过程，我们只需要调用方法 TransformDirection()，就可以变换方向。

定义　在这个上下文中，Transform(变换)意味着从一个坐标空间转换为另一个坐标空间(如果不记得什么是坐标空间，请参阅 2.3.3 节)。不要与变换的其他定义混淆起来，包括 Transform 组件和在场景中移动对象这个行为。它们转义了这个术语，因为所有这些都指向同一个基本概念。

现在尝试运行移动代码。如果还没这么做，将 MouseLook 组件设置为同时水平和垂直旋转。可以通过键盘控制浏览整个场景，并在场景中飞来飞去。如果想让玩家能在场景中飞翔，这确实不错，但如果想让玩家只能在地面行走，该怎么办呢？

2.5.4　将组件调整为走路而不是飞翔

现在碰撞检测已经奏效，脚本可以添加重力设置，让玩家一直停留在地面上。声明 gravity 变量，给 Y 轴使用这个值，如代码清单 2.12 所示。

代码清单 2.12　将重力添加到移动代码

```
...
public float gravity = -9.8f;
...
void Update(){
  ...
  movement = Vector3.ClampMagnitude(movement, speed);
  movement.y = gravity;
  ...
```

使用重力值而不只是 0

现在玩家承受一个固定向下的力，但它不是相对于玩家永远竖直向下的，因为玩家对象能通过鼠标上下倾斜。幸运的是，我们需要修复的对象已经有了，因此只需要对玩家身上组件的设置进行些许调整即可。首先将玩家对象的 MouseLook 设置为仅水平旋转。接着给摄像机对象添加一个 MouseLook 组件，并将它设置为仅垂直旋转。现在，就有了两个响应鼠标的不同对象！

因为玩家对象现在只能水平旋转，所以竖直向下的重力不再会被倾斜。摄像机对象的父对象是玩家(还记得在 Hierarchy 视图中的操作吗？)，所以摄像机尽管独立于玩家做垂直旋转，它还是会跟着玩家做水平旋转。

打磨已完成的脚本

使用 RequireComponent()方法确保脚本附加了其他需要的组件。有时一些组件是可选的(也就是，代码指明“如果附加了这个组件，则......”)，但一些组件是必选的。在脚本的顶部添加 RequireComponent()方法来实现必选项，把需要的组件作为参数。

与此类似，如果将方法 AddComponentMenu()添加到脚本的顶部，脚本将添加到 Unity 编辑器的组件菜单中。将想添加的菜单项名称告诉命令，那么当单击 Inspector 底部的 Add Component 时就能选择这个脚本。太简单了！

将两个方法添加到顶部的脚本如下所示：

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))]
[AddComponentMenu("Control Script/FPS Input")]
public class FPSInput : MonoBehaviour {
...
```

代码清单 2.13 展示了全部完成后的脚本。调整玩家上的组件设置，玩家就能在房间中行走。即使应用了 gravity 变量，依然可以在 Inspector 中把 gravity 设置为 0，使用这个脚本让玩家角色飞起来。

代码清单 2.13　完成的 FPSInput 脚本

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))]
[AddComponentMenu("Control Script/FPS Input")]
public class FPSInput : MonoBehaviour {
 public float speed = 6.0f;
 public float gravity = -9.8f;

 private CharacterController _charController;

 void Start() {
 _charController = GetComponent<CharacterController> ();
 }

 void Update() {
   float deltaX = Input.GetAxis("Horizontal") * speed;
   float deltaZ = Input.GetAxis("Vertical") * speed;
   Vector3 movement = new Vector3 (deltaX, 0, deltaZ);
   movement = Vector3.ClampMagnitude (movement, speed);

   movement.y = gravity;

   movement *= Time.deltaTime;
   movement = transform.TransformDirection (movement);
   _charController.Move (movement);
 }
}
```

恭喜构建了这个 3D 项目！本章打下了一些基础，现在我们知道如何在 Unity 中编写移动代码。尽管第一个演示游戏令人激动，但它要成为一个完整的游戏，还有很长的路要走。毕竟，项目计划中把这个游戏描述为基础的 FPS 场景，而如果不能射击，又

怎么称得上是射击者？所以在本章项目之后适当鼓励一下自己，并准备进行下一步。

2.6　小结

- 3D 坐标空间用 X，Y，Z 轴定义。
- 房间中的对象和光源构成场景。
- 第一人称场景中的玩家本质上是一个摄像机。
- 移动代码不停地在每帧应用小的变换。
- FPS 控制由鼠标旋转和键盘移动构成。

第 3 章

为 3D 游戏添加敌人和子弹

本章涵盖：

- 讨论瞄准和开火，同时适用于玩家和敌人
- 碰撞检测和反馈
- 让敌人四处走动
- 在场景中生成新对象

第 2 章的移动演示游戏很酷，但依然不是一款真正的游戏。下面将该移动演示游戏变成第一人称射击游戏。如果现在考虑一下还需要什么，则显然需要射击的能力和可以射击的东西。首先编写脚本，使玩家能在场景中向对象射击。接着构建敌人来填充场景，包括漫无目的徘徊的敌人和对受击做出反应的代码。最后允许敌人回击，向玩家发射火球。第 2 章的脚本不需要修改，本章给该项目添加新脚本，以处理新增的特性。

这个项目选择第一人称射击有一些原因。一个简单的原因是 FPS 游戏较流行，人们喜欢射击游戏，所以本书制作射击游戏。另一个精妙的原因是与要学习的技术有关，这个项目是学习 3D 仿真中几个基本概念的一种绝佳方式。例如，射击游戏是学习射线发射的绝佳方式。稍后讲解射

线发射的细节，但现在只需要知道，射线发射适合于在 3D 仿真中实现不同的任务。尽管射线发射在各种情况中都能派上用场，但最直观的感觉是，它能用于射击。

为了创建用于射击的徘徊目标，需要探索计算机控制的角色的代码，和发送消息并生成对象所使用的技术。实际上，这个徘徊的行为是另一个使用射线发射的地方，所以在首次通过射击学习射线发射之后，就关注这项技术的另一个应用。类似的，项目中演示的消息发送方法也可以应用于其他地方。后续章节将介绍这些技术的其他应用，甚至在这个项目中也将介绍其他的应用情况。

最后，这个项目每次只实现一个特性，使游戏在每一步都可玩，而不是总觉得少做了一些工作。路线图将步骤分解为多个可理解的小步骤，每步仅添加一个新特性。

(1) 编写代码，允许玩家向场景射击。

(2) 创建能响应被击中的静态目标。

(3) 让目标四处走动。

(4) 自动产生四处走动的目标。

(5) 让目标/敌人向玩家发射火球。

注意 本章的项目假设已经构建了第一人称的移动演示游戏。第 2 章创建了移动的演示游戏，但如果跳过了第 2 章，就需要下载该章的示例文件。

3.1 通过射线射击

3D 演示游戏中第一个要介绍的新特性是射击。四处查看并移动肯定是第一人称射击游戏中的重要特性，但仅当玩家能影响仿真并应用其技能时，它才是游戏。3D 游戏中的射击可以通过几种不同的方法实现，但最重要的方法是射线发射。

3.1.1 什么是射线发射

顾名思义，射线发射就是向场景发射一条射线。很清楚，是吧？那么射线究竟是什么？

定义 射线是场景中虚拟的或看不见的线，它从原点开始，沿着指定的方向延伸出去。

创建一条射线，并判断它和什么对象相交，这就是射线发射；图 3-1 阐述了这个概念。考虑从枪中发射子弹的情景：子弹从枪口发出，沿着直线向前飞行，直到它撞到某个东西。射线类似子弹的路径，射线发射则模拟发射子弹，看它会碰撞到何物。

可以想象，射线发射背后的数学通常很复杂。不只是射线和 3D 平面的交点的计算很棘手，还需要对场景中所有网格对象的所有多边形进行计算(记住，网格对象是由一些连线和形状构成的 3D 可视化结构)。幸运的是，Unity 处理了射线发射背后复杂

的数学，但我们依然需要了解高级概念，例如射线从哪里开始和为什么发射。

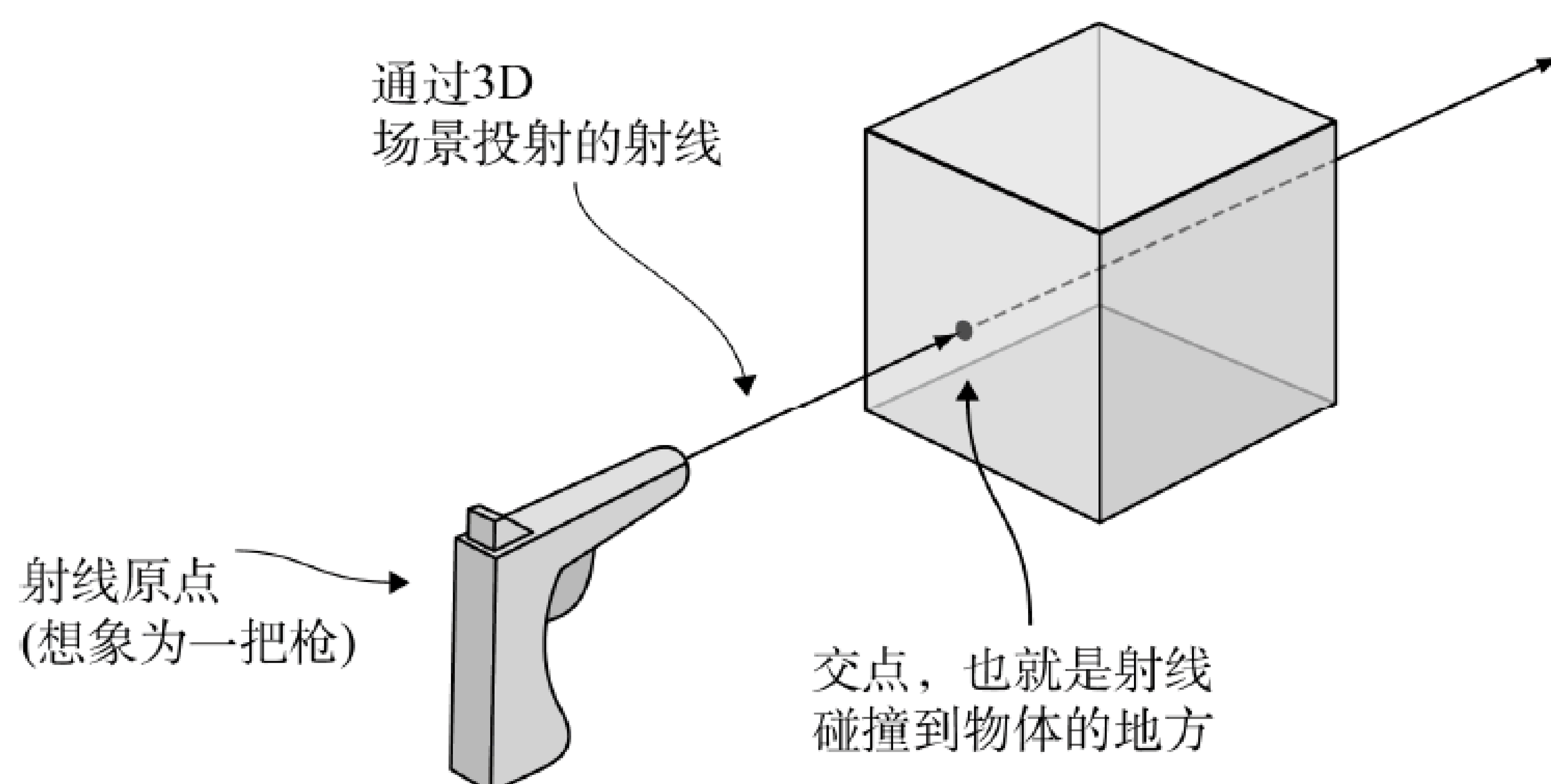


图 3-1　射线是虚拟的线，射线发射是找到该线在哪里与某物相交

在这个项目中，后者(为什么发射)的答案是模拟子弹射向场景。对于第一人称射击游戏而言，射线通常开始于摄像机位置，并通过摄像机视图中心往外延伸。换句话说，就是检查摄像机正前方的对象；Unity 提供的命令可以简化这个任务。下面介绍这些命令。

3.1.2　使用命令 ScreenPointToRay 射击

下面在实现射击时，会投射一条射线，该射线源于摄像机，并通过摄像机视图中心往前方延伸。通过摄像机视图中心投射一条射线是所谓“鼠标拾取(mouse picking)”操作的一个特例。

定义　鼠标拾取是在 3D 场景中选中鼠标光标所指向的对象的操作。

Unity 提供了 ScreenPointToRay()方法来执行这个操作。图 3-2 阐述了调用该方法所执行的操作。该方法创建一个从摄像机开始的射线，并射向给定的屏幕坐标。通常，鼠标拾取使用的是鼠标位置的坐标，但对于第一人称射击游戏，则使用屏幕中心。一旦有了射线，它就能传入 Physics.Raycast()方法，从而使用该射线执行射线发射。

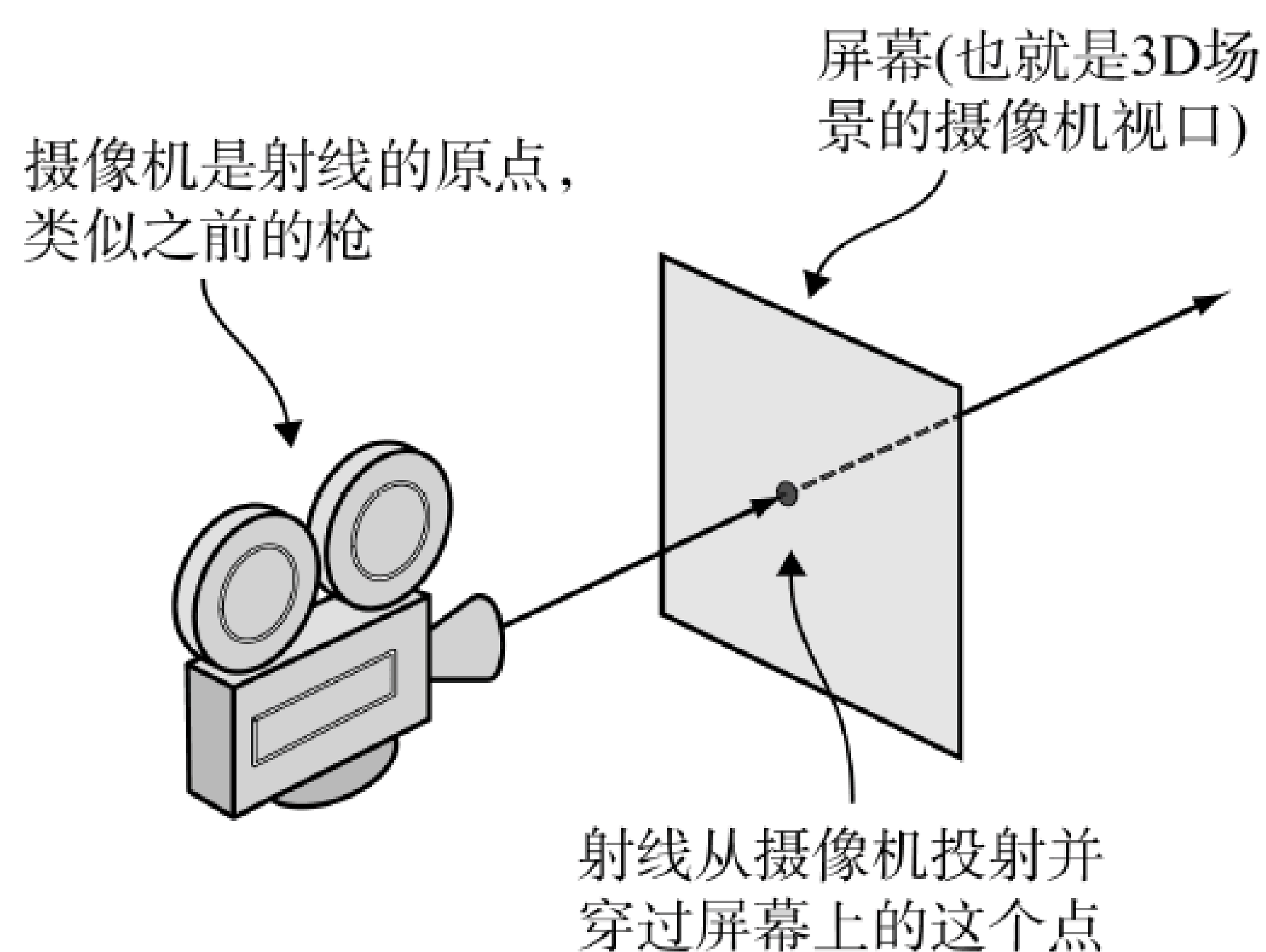


图 3-2　ScreenPointToRay()从摄像机投射射线，穿过给定的屏幕坐标

下面使用刚刚讨论的方法编写代码。在 Unity 中创建新的 C#脚本，命名为 RayShooter。把脚本附加到摄像机上(不是玩家对象)，然后在其中输入代码清单 3.1 所示的代码。

代码清单 3.1　附加到摄像机上的 RayShooter 脚本

```
using UnityEngine;
using System.Collections;

public class RayShooter : MonoBehaviour {
  private Camera _camera;

  void Start() {
    _camera = GetComponent<Camera>();
  }

  void Update() {
    if (Input.GetMouseButtonDown (0)) {
      Vector3 point = new Vector3(_camera.pixelWidth/2, _camera.
     pixelHeight/2, 0);
      Ray ray = _camera.ScreenPointToRay(point);
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit)) {
        Debug.Log("Hit "+hit.point);
      }
    }
  }
}
```

访问相同对象上附加的其他组件

响应鼠标按键

raycast 给引用的变量填充信息

检索射线击中的坐标

使用 ScreenPointToRay()在摄像机所在位置创建射线

屏幕中心是宽高的一半

在这个代码清单中，应该注意一些事项。首先，Camera 组件在 Start()中检索，就像之前章节中的 CharacterController 一样。接着剩下的代码放到 Update()中，因为需要重复检查鼠标，而不是只检查一次。Input.GetMouseButtonDown()方法是返回 true 还是 false，取决于是否单击了鼠标，因此把 Input.GetMouseButtonDown()放到一个条件中，这意味着只有单击了鼠标，才运行其中的代码。由于玩家单击鼠标是为了射击，因此执行该条件，检查鼠标按钮是否被按下。

创建向量是为了定义射线的屏幕坐标(记住向量把几个相关的数字保存在一起)。摄像机的 pixelWidth 和 pixelHeight 值指定了屏幕的大小，因此将这两个值除以 2 可以获得屏幕的中心。尽管屏幕坐标是二维的，只有水平和垂直分量，而没有深度，但依然要创建三维向量 Vector3，因为 ScreenPointToRay()需要 Vector3 数据类型(推测原因为射线的计算涉及 3D 向量的算术)。ScreenPointToRay()使用传入的坐标来调用，产生一个 Ray 对象(代码对象，而不是游戏对象；有时会混淆这两者)。

接着射线传入 Raycast()方法，但它不是传入的唯一对象，还传入了 RaycastHit 数据结构。RaycastHit 是关于射线交叉的一组信息，包括在哪里交叉和与哪个对象发生交叉。C#语法 out 确保在命令内操作的数据结构就是命令外部的同一个对象，这和那

些在不同函数作用域中复制的对象相反。

有了这些参数，Physics.Raycast()方法就可以工作了。这个方法检测与给定射线的交叉，把交叉信息填充到 data 中，并且在碰撞到任何事物时返回 true。因为返回的是布尔值，所以这个方法可以放到条件检查语句中，就像前面使用 Input.GetMouseButtonDown()时那样。

现在代码发出一个控制台消息，表明交叉何时发生。控制台消息显示射线击中点的 3D 坐标(第 2 章讨论的 XYZ 值)。但这很难形象地表示射线具体击中了何处，同样，现在也很难确定屏幕的中心(即射线穿过的地方)在何处。下面添加可视化指示器，来处理这两个问题。

3.1.3　为准心和击中点添加可视化指示器

下一步是添加两种类型的可视化指示器：在屏幕中心的准心和场景中射线碰撞的位置标记。对于第一人称射击游戏，后者通常是弹孔，但现在只要放一个空球体在该点即可(并使用一个协程在 1 秒后移除球体)。图 3-3 展示了结果。

定义　协程(coroutines)是 Unity 处理任务的特有方式，这些任务随着时间的推移逐步执行，这种方式与大多数函数让程序等待直到它们完成相反。

首先，添加指示器来标记射线碰撞到何处。代码清单 3.2 展示了完成这个添加操作之后的代码。在场景中四处开枪，将看到很有趣的球体指示器！

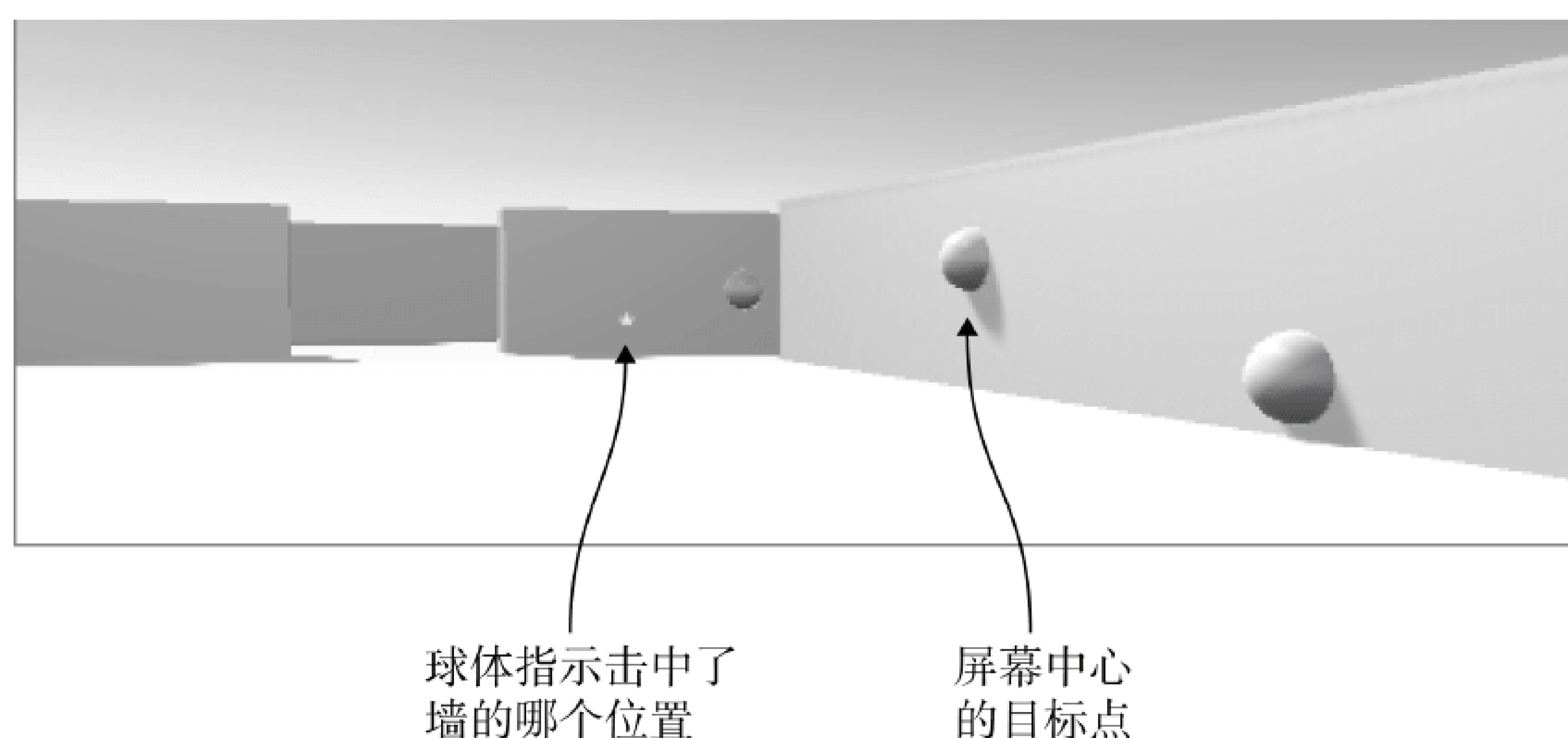


图 3-3　在为准心和击中点添加可视化指示器之后重复射击

代码清单 3.2　添加了球体指示器的 RayShooter 脚本

```
using UnityEngine;
using System.Collections;

public class RayShooter : MonoBehaviour {
  private Camera _camera;

  void Start() {
    _camera = GetComponent<Camera>();
```

```
    }

    void Update() {
      if (Input.GetMouseButtonDown (0)) {
        Vector3 point = new Vector3(_camera.pixelWidth/2, camera.
      pixelHeight/2, 0);
        Ray ray = _camera.ScreenPointToRay(point);
        RaycastHit hit;
        if (Physics.Raycast(ray, out hit)) {
          StartCoroutine(SphereIndicator(hit.point));
        }
      }
    }

    private IEnumerator SphereIndicator(Vector3 pos){
      GameObject sphere = GameObject.CreatePrimitive (PrimitiveType.Sphere);
      sphere.transform.position = pos;

        yield return new WaitForSeconds (1);

        Destroy (sphere);
    }
}
```

这个方法的大部分代码与代码清单 3.1 中的射线发射代码一样

运行协程来响应击中

协程使用 IEnumerator 方法

yield 关键字告诉协程在何处暂停

移除 GameObject 并清除它占用的内存

在此添加了新方法 SphereIndicator()，并在已有的 Update()方法中修改了一行代码。这个方法在场景中的一个点创建球体，接着在随后 1 秒移除它。在射线发射代码中调用 SphereIndicator()，可以确保可视化指示器能精确显示射线击中的位置。SphereIndicator()方法使用 IEnumerator 定义，IEnumerator 类型和协程的概念相关。

从技术上讲，协程不是异步的(异步操作不会停止正在运行的其他代码，回顾一下在网站的脚本中下载图片)，但通过巧妙利用枚举，Unity 使协程变得很像异步方法。协程的秘密在于 yield 关键字，这个关键字会使协程临时暂停，挂起程序流并在下一帧继续运行。通过这种方式，重复运行部分程序，并返回程序的剩下部分，使协程看起来是在程序后台运行。

顾名思义，StartCoroutine()启动一个协程。一旦协程启动，它就会一直运行，直到函数结束；它只是在运行过程中暂停。注意这个微妙的要点是，传入 StartCoroutine()的方法名称后面跟着一对括号，而不是只传入它的名称：这个语法意味着调用该函数。被调用的函数一直运行，直到它遇到 yield 命令，则函数暂停。

SphereIndicator()在指定点创建一个球体，在遇到 yield 语句时暂停，接着在协程恢复之后销毁球体。暂停的时间长度取决于 yield 返回的值。协程上可以使用一些不同类型的返回值，但最直接的方式是返回指定的等待时长。返回 WaitForSeconds(1)则表示协程暂停 1 秒。创建球体，暂停 1 秒，然后销毁球体：SphereIndicator()方法中的代码序列建立了一个临时的可视化指示器。

代码清单 3.2 标记了射线碰撞位置的指示器。但还需要屏幕中心的准心，这正是

代码清单 3.3 的作用所在。

代码清单 3.3　用于准心的可视化指示器

```
...
void Start() {
  _camera = GetComponent<Camera>();

  Cursor.lockState = CursorLockMode.Locked;
  Cusor.visible = false;
}

void OnGUI() {
  int size = 12;
  float posX = _camera.pixelWidth/2 - size/4;
  float posY = _camera.pixelHeight/2 - size/2;
  GUI.Label(new Rect(posX, posY, size, size), "*");
}
...
```

隐藏屏幕中心的光标

GUI.Label()命令在屏幕上显示文本

添加到 RayShooter 类中的另一个新方法为 OnGUI()。Unity 既有基础的也有高级的 UI 系统，由于基础的 UI 系统存在很多限制，因此后续章节将使用更灵活的高级 UI 系统，但现在使用基础 UI 能更容易地在屏幕中心显示一个点。和 Start()和 Update()方法非常类似，每个 MonoBehaviour 都自动响应 OnGUI 方法。OnGUI 方法在每帧 3D 场景被渲染之后执行，它可以使 OnGUI()中绘制的东西出现在 3D 场景最上层(想象一下将贴纸贴到风景画上的情景)。

定义　渲染是计算机绘制 3D 场景的像素的操作。虽然场景使用 XYZ 坐标定义，但真正显示在显示器上的是彩色像素的 2D 网格。因此为了显示 3D 场景，计算机需要在 2D 网格中计算所有像素的颜色，运行的这种算法称为渲染。

在 OnGUI()代码内定义了用于显示的 2D 坐标(由于文本的大小，轻微做了偏移)并调用了 GUI.Label()。GUI.Label()方法显示文本标签；因为传入 label 中的字符串是星号(*)，所以最终会在屏幕中心看到星号字符。现在在这个初生的 FPS 游戏中更容易瞄准！

代码清单 3.3 也在 Start()方法中添加了一些光标设置，为光标的可见性和锁定设置值。如果忽略光标的值，这个脚本也能完美工作，但是这些设置使第一人称射击游戏的控件工作得更顺畅。鼠标光标一直停留在屏幕中心，而为了避免造成混乱，它一直不显示，只有按下 Esc 键，图才会重新出现。

警告　永远要记住，可以按 Esc 键解锁鼠标光标。当锁定鼠标光标时，就无法单击 Play 按钮，停止游戏。

上面完成了第一人称射击游戏的代码，完成了玩家交互的收尾工作，接下来还需要关注射击的目标。

3.2　编写能响应的目标

玩家在游戏中可以射击了，但现在没有东西可供玩家射击。下面创建一个目标对象，并编写一个脚本，来响应受击。更精确地说，我们将稍微修改射击代码，在目标被击中时通知它，接着附加在目标上的脚本将在收到通知时做出反应。

3.2.1　确定被击中的对象

首先需要创建一个用于射击的新对象。创建一个新立方体对象(GameObject | 3D Object | Cube)，然后将 Y 比例设置为 2，将 X 和 Z 比例设置为 1，在垂直方向上拉伸它。把新对象定位在(0，1，0)，将它放在房间中间的地面上，并将该对象命名为 Enemy。创建一个新的脚本 ReactiveTarget，并将其附加到新建的立方体上。后面很快就为这个脚本编写代码，但现在保持默认状态。现在仅需要创建这个脚本文件，是因为接下来的代码清单 3.4 需要它存在才能编译。回到 RayShooter.cs 并按照代码清单 3.4 修改射线发射的代码。运行新代码，并向新目标射击；调试消息会显示在控制台上，而不是在场景中出现球体指示器。

代码清单 3.4　检测是否击中目标对象

```
...
if (Physics.Raycast(ray, out hit)) {
  GameObject hitObject = hit.transform.gameObject;
  ReactiveTarget target = hitObject.GetComponent<ReactiveTarget>();
  if (target != null) {
    Debug.Log("Target hit");
  } else {
    StartCoroutine(SphereIndicator(hit.point));
  }
}
...
```

检索射线击中的对象

检查对象上是否有 ReactiveTarget 组件

注意从 RaycastHit 中检索所击中的对象，就像为球体指示器获取坐标一样。从技术上讲，击中信息不会返回所击中的对象，它指明了所击中的 Transform 组件。可以通过 transform 的属性访问 gameObject。

接着，使用对象的 GetComponent()方法来检查它是不是一个有响应的目标(如果附加了 ReactiveTarget 脚本就是)。如前所述，GetComponent()方法返回附加到 GameObject 上的指定类型的组件。如果对象上没有添加这个类型的组件，则 GetComponent()不会返回任何东西。可以检查 GetComponent()是否返回 null 并在每个分支执行不同代码。

如果击中的对象是一个有响应的目标，代码就发送一个调试消息，而不是启动球

体指示器的协程。现在告知目标对象，它已被击中，以便它做出反应。

3.2.2　警告目标被击中

为此，只需要改变一行代码即可，如代码清单 3.5 所示。

代码清单 3.5　将消息发送给目标对象

```
...
if (target != null) {
  target.ReactToHit();          // 调用target的方法而不仅仅是发送调试消息
} else {
  StartCoroutine(SphereIndicator(hit.point));
}
...
```

现在射击代码调用目标的方法，因此需要编写该目标的方法。在 ReactiveTarget 脚本中，编写代码清单 3.6 中的代码。当射到目标对象时，它将跌倒并消失。如图 3-4 所示。

代码清单 3.6　当受击时 ReactiveTarget 脚本终止

```
using UnityEngine;
using System.Collections;

public class ReactiveTarget : MonoBehaviour {

  public void ReactToHit() {          // 通过射击脚本调用的方法
    StartCoroutine(Die());
  }

  private IEnumerator Die() {          // 推倒敌人，等待 1.5 秒后销毁敌人
    this.transform.Rotate(-75, 0, 0);

    yield return new WaitForSeconds(1.5f);

    Destroy(this.gameObject);          // 对象能销毁自己，就像一个独立的对象
  }
}
```

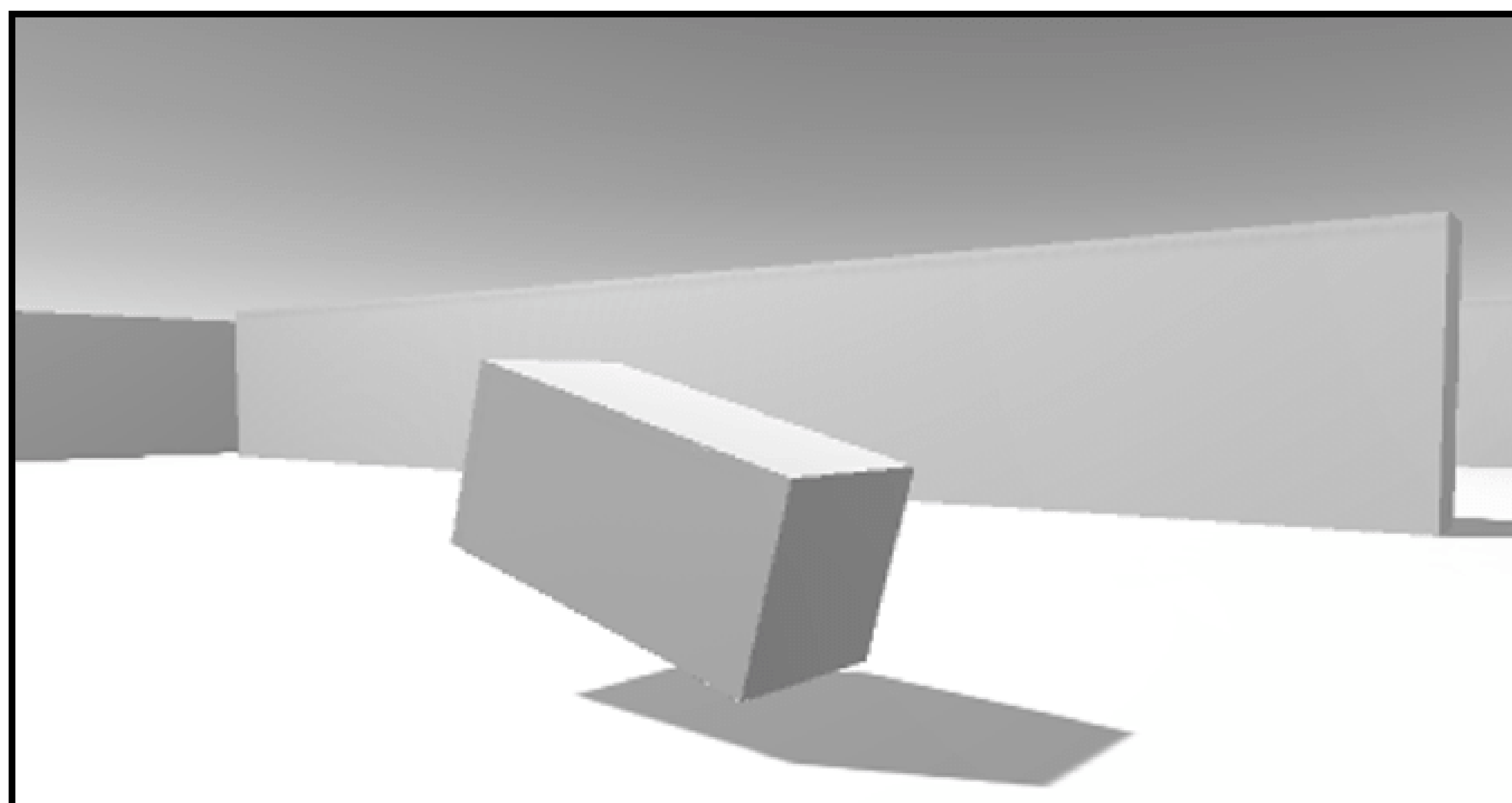


图 3-4　当目标对象受击时倾倒

这段代码的大部分内容都应该很熟悉，就像之前的脚本一样，因此只需要浏览一下。首先定义方法 ReactToHit()，因为该方法名在射击脚本中调用。ReactToHit()方法启动了一个协程，该协程类似于之前球体指示器的代码，其主要区别在于该方法操作这个脚本的对象，而不是创建一个独立的对象。类似 this.gameObject 这样的表达式指向脚本附加的 GameObject (this 关键字是可选的，因此代码没有 gameObject 前面的部分也能引用 gameObject)。

协程函数 Die()的第一行让对象倾斜。如第 2 章所述，可以将旋转定义为绕三个坐标轴 X、Y 和 Z 旋转的角度。因为对象不应从一边旋转到另一边，因此让 Y 和 Z 的旋转角度为 0，而将 X 轴的旋转角度设置一个值。

注意　这个变换是即刻生效的，但当对象倾倒时是运动的。一旦开始为了一些高级话题拓展本书视野，就可能需要查阅缓动(tweens)，即使对象随着时间平滑移动的系统。

Die()方法的第二行使用了 yield 关键字，它是协程暂停函数的关键，而且它返回在恢复前需要等待的秒数。最后，游戏对象在 Die()方法的最后一行销毁自己。Destroy(this.gameObject)在等待时间后调用，就像之前代码调用 Destroy(sphere)那样。

警告　确认对 this.gameObject 调用的是 Destroy 而不是简单的 this！不要混淆这两者；this 只是指向这个脚本组件，然而 this.gameObject 指向脚本所附着的对象。

现在目标响应了受击；很好！但它什么都没有做，因此下面添加更多的操作，让这个目标成为更恰当的敌人角色。

3.3　基本漫游 AI

静止的目标一点趣味也没有，因此下面编写代码让敌人到处闲逛。四处走动的代码是 AI 最简单的例子；人工智能(AI)指的是计算机控制的实体。在这个例子中，实体就是游戏中的敌人，它也可以是现实世界的机器人，或一个下棋的声音等。

3.3.1　图解基础 AI 的工作原理

实现 AI 有很多不同的方法(严格来讲，AI 是计算机科学一个重要的研究领域)，但就这里的目的而言，我们将采用简单的方法。随着读者的经验越来越丰富，游戏越来越复杂，就可能想研究实现 AI 的不同方法。

图 3-5 描述了基础流程。在每帧，AI 代码将扫描它的环境，来决定它是否需要响应。如果敌人的路上出现障碍物，敌人就会转向另一个方向。不管敌人是否需要

转向，他一直会往前走。因此敌人在房间里面来回走，一直往前移动，并在遇到墙壁时转向。

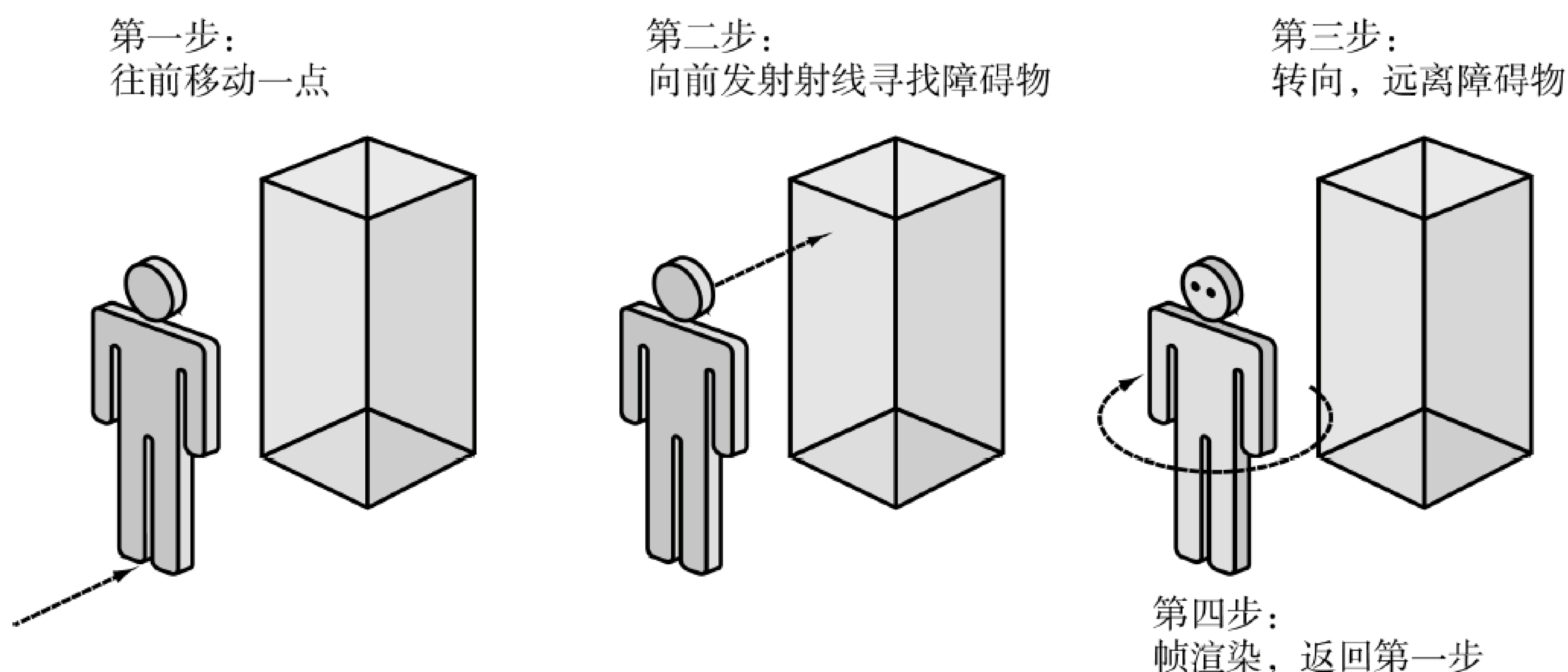


图 3-5　基础 AI：周期性地处理向前移动和躲避障碍物

实际代码看起来很熟悉，因为向前移动敌人所使用的命令和向前移动玩家相同。AI 代码也使用了射线发射，和射击代码很像，但是在不同的情景下使用。

3.3.2　使用射线发射发现障碍物

如前所述，射线发射是用于处理 3D 仿真中一些任务的一项技术。一个易于理解的任务就是射击，但射线发射的另一个用途还包括扫描场景。使用射线发射扫描场景是 AI 代码中的一个步骤，这意味着 AI 代码使用了射线发射。

前面在摄像机位置创建射线，因为那是玩家面向场景的位置；这次将创建一个源于敌人的射线。第一条射线从摄像机射出并经过屏幕中心，因为这次射线将往角色前方发射，如图 3-6 所示。接着就像射击代码中使用 RaycastHit 信息决定是否有东西被击中和在哪里击中一样，AI 代码也使用 RaycastHit 信息来判断敌人前面是否有什么东西，如果有，则判断距离有多远。

射击的射线发射和 AI 的射线发射间的一个区别是要检测的射线半径。对于射击，射线的半径是无限小的，而 AI 的射线有一个相当大的截面，这意味着代码使用的是 SphereCast()，而不是 Raycast()。这两者不同的原因是子弹很小，而检查角色前面的障碍物需要考虑角色的宽度。

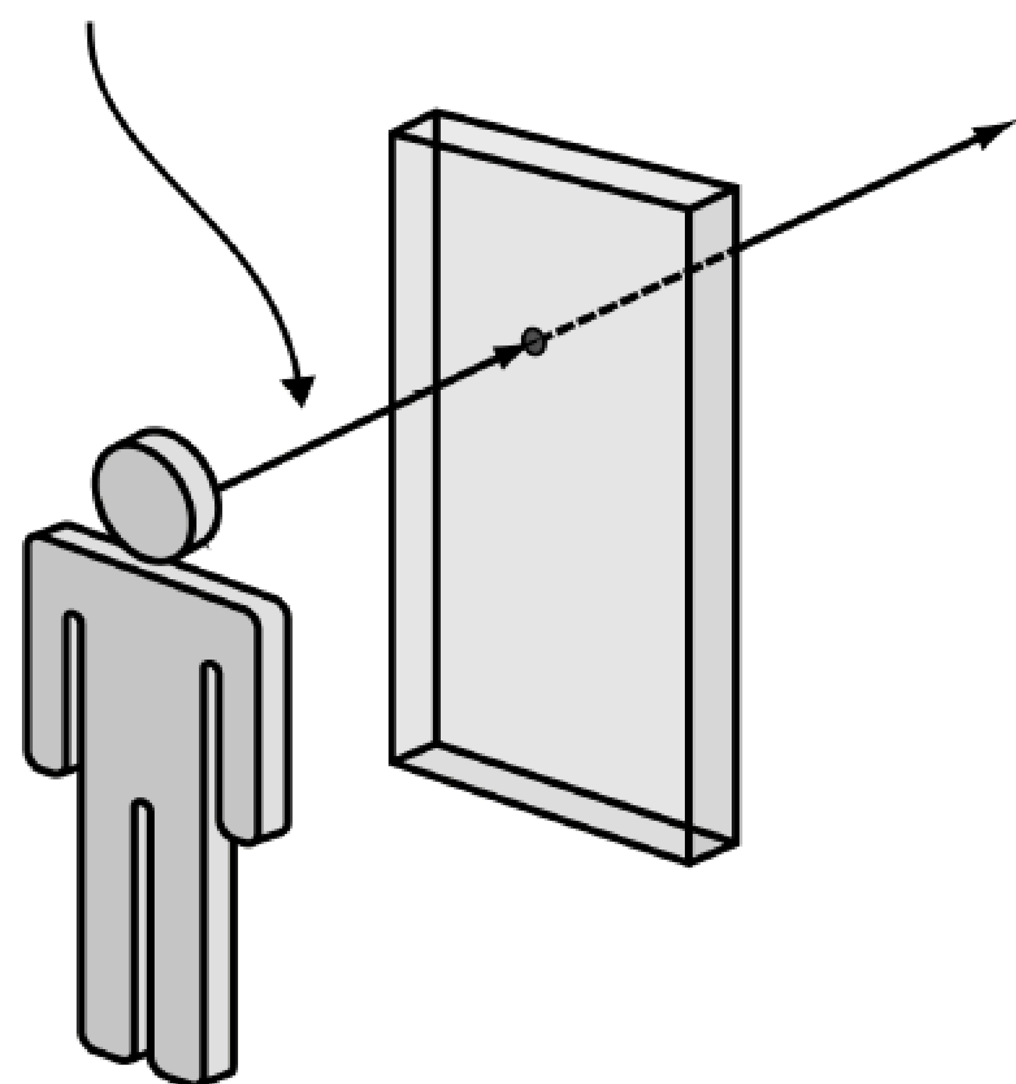


图 3-6　使用射线发射来查找障碍物

创建一个新脚本 WanderingAI，将其附加到目标对象上(在 ReactiveTarget 脚本旁边)，并编写代码清单 3.7 所示的代码。现在运行场景，就应看到敌人在房间中徘徊；我们依然能射击目标，目标会以之前的方式做出反应。

代码清单 3.7　基本的 WanderingAI 脚本

```
using UnityEngine;
using System.Collections;

public class WanderingAI : MonoBehaviour {
  public float speed = 3.0f;
  public float obstacleRange = 5.0f;

  void Update() {
    transform.Translate(0, 0, speed * Time.deltaTime);

    Ray ray = new Ray(transform.position, transform.forward);
    RaycastHit hit;
    if (Physics.SphereCast(ray,0.75f, out hit)) {
      if (hit.distance<obstacleRange) {
        float angle = Random.Range(-110, 110);
        transform.Rotate(0, angle, 0);
      }
    }
  }
}
```

移动速度的值和对障碍物做出反应的距离

每帧持续向前移动，不管转向

和角色相同位置的射线，并且方向相同

使用沿着射线的周长发射射线

转向一个半随机的新方向

代码清单 3.7 添加了一些变量来表示移动速度和 AI 对障碍做出反应的距离。接着将 Translate()方法添加到 Update()方法中，用于持续向前移动(包括使用 deltaTime 使移动独立于帧率)。在 Update()中，射线发射的代码看起来和之前射击脚本中的一样；此外，这里也使用射线发射技术而不是射击来查找障碍物。射线使用敌人的位置和方向而不是摄像机创建。

如前所述，射线发射的计算使用方法 Physics.SphereCast()来处理。这个方法使用半径参数来决定在多大范围内进行碰撞检测，但在其他方面，它和 Physics.Raycast()一样。这个相似性包括了命令如何填充碰撞信息，像之前一样检查碰撞，使用 distance 属性确保当玩家和障碍物比较近时才做出反应(而不是穿越房间里的墙)。

当敌人前面有靠近的障碍物时，代码将角色旋转到一个半随机的新方向。这里所说的“半随机”是因为旋转角度在最大值和最小值之间。具体而言，使用 Unity 提供的方法 Random.Range()来获取最大值和最小值之间的一个随机数。在这个例子中，约束只是稍微往左转或往右转，允许角色充分旋转来避免障碍物。

3.3.3　跟踪角色的状态

当前行为有个奇怪之处是当敌人受击跌倒后，还继续往前移动。这是因为 Translate()方法不管什么情况都会每帧运行。下面对代码做一些小调整，以跟踪角色

是否还存活——用另一种技术术语来表达，就是追踪角色的“alive”状态。让代码不断跟踪对象的状态，根据对象的当前状态做出不同的反应，这是编程随处可见的一种代码模式，而不仅是在 AI 中。这种方法的更复杂的实现称为状态机，或称为有限状态机。

定义 有限状态机(finite state machine，FSM)是一种代码结构，用于跟踪对象的当前状态。状态间存在明确定义的转换，且代码基于状态而有所不同。

我们不会实现完整的 FSM，但很多讨论 AI 的地方经常首先讲到 FSM，这绝非偶然。完整的 FSM 会有很多状态用于复杂 AI 中的所有不同行为。但在这个基础 AI 中，只需要追踪角色是否活着。代码清单 3.8 在脚本顶部添加了布尔值_alive，代码需要对该值做条件检查。有了这些检查，移动代码只会在敌人还活着时运行。

代码清单 3.8 添加“alive”状态的 WanderingAI 脚本

```
...
private bool _alive;                    ← 布尔值用于跟踪敌人是否存活

void Start() {
  _alive = true;                        ← 初始化该值
}

void Update() {
  if (_alive) {                         ← 只有当角色存活时才移动
    transform.Translate(0, 0, speed*Time.deltaTime);
    ...
  }
}
                                        ← 公有方法允许外部代码影响“alive”的值
public void SetAlive(bool alive) {
  _alive = alive;
}
...
```

现在 ReactiveTarget 脚本能告诉 WanderingAI 脚本敌人是否存活(查看代码清单 3.9)。

代码清单 3.9 ReactiveTarget 告诉 WanderingAI 什么时候死亡

```
...
public void ReactToHit() {
    WanderingAI behavior = GetComponent<WanderingAI>();
    if (behavior != null) {             ← 检查角色是否有 WanderingAI 脚本；可能没有
        behavior.SetAlive(false);
    }
    StartCoroutine(Die());
}
...
```

AI 代码结构

本章的 AI 代码包含在一个类中，所以学习和理解它很简单。对于简单的 AI 需求，

这个代码结构非常合适，所以不要害怕做“错”事，更复杂的代码结构是绝对需要的。对于更复杂的 AI 需求(例如，游戏有一些高智能角色)，更健壮的代码结构有助于更容易地开发 AI。

正如第 1 章示例中提及的组合和继承，有时想把 AI 切分到独立的脚本中。这样就可以组合和搭配组件，为每个角色生成独一无二的行为。思考角色的相同点和不同点，设计代码的架构时，这些不同点将提供灵感。例如，如果游戏中的一些敌人轻率地冲向玩家，一些敌人在暗处潜行，就可以让 Locomotion 成为一个独立的组件。接着就可以为 LocomotionCharge 和 LocomotionSlink 创建脚本，给不同的敌人使用不同的 Locomotion 组件。

AI 代码结构取决于特定游戏的设计，实现它没有所谓“正确”的方法。Unity 使设计灵活的代码架构更容易。

3.4 生成敌人预设

此时，场景中只有一个敌人，当它死后，场景就空了。下面让游戏产生敌人，这样无论某个敌人什么时候死亡，就会出现新的敌人。在 Unity 中使用称为预设(prefab)的概念很容易实现这一点。

3.4.1 什么是预设

预设是一种可视化地定义交互对象的灵活方法。简言之，预设是完全具象化的游戏对象(已经附加了脚本和设置好的组件)，它不存在于任何特定场景，却能作为资源存在，能复制到任何场景中。这个复制操作能手动处理，以确保敌人对象(或其他预设)在每个场景中都一样。更重要的是，预设也能从代码中产生；不仅能使用脚本中的命令在场景中放置对象的副本，也能在可视化编辑器中手动完成。

定义 asset(资源)是 Project 视图中显示的任意文件，它们可以是 2D 图像、3D 模型、代码文件、场景等。在第 1 章简要提过 asset，但直到现在才详述它。

预设的副本称为实例，类似于从类中创建的特定代码对象。要尽量让术语清晰，预设是指存在任何场景外的游戏对象，而实例指放到场景中的对象副本。

定义 类似面向对象术语，实例化是指创建实例的这一操作。

3.4.2 创建敌人预设

为了创建预设，首先在场景中创建一个要变成预设的对象。因为敌人对象将变成

预设，所以第一步已经完成了。现在只需要将对象从 Hierarchy 视图拖放到 Project 视图中；这将自动把对象保存为预设(见图 3-7)。回到 Hierarchy 视图，源对象的名称将变成蓝色，这意味着它现在链接到一个预设。如果以后想编辑这个预设(例如添加新组件)，只需要在场景上对这个对象做些修改，然后选择 GameObject | Apply Changes To Prefab。但这个对象不应继续留在场景中(我们将产生预设，不使用已经在场景中的预设)，因此现在删除敌人对象。



图 3-7　将对象从 Hierarchy 视图拖动到 Project 视图来创建预设

警告 用于操作预设的界面有点简陋，而预设和场景中它们的实例的关系有点脆弱。例如，通常只能将一个预设拖到场景中编辑它，然后在完成编辑之后删除它。在第 1 章中提过这是 Unity 的缺点，希望 Unity 的后续版本能改善预设的工作流。

现在预设对象是在场景中产生的，因此下面编写代码来创建预设的实例。

3.4.3　在不可见的 SceneController 中实例化

虽然预设自身不存在于场景中，但场景中需要有一些对象来附加产生敌人的代码。为此需要创建一个空的游戏对象。可以将脚本附加到它上面，但这个对象在场景中是不可见的。

提示 使用空 GameObject 来附加脚本组件是 Unity 开发中的一种常见模式。这个窍门用于那种不应用到场景中任何特定对象上的抽象任务。Unity 脚本意在附加到可见对象上，但不是每个任务那样做都有意义。

选择 GameObject | Create Empty，将新对象重命名为 Controller，接着将它的位置设置为(0,0,0)(从技术上来说，这个位置不重要，因为对象不可见，但是把它放在原点会在让它成为其他东西的父节点时变得更简单)。创建脚本 SceneController，如代码清单 3.10 所示。

代码清单 3.10　产生敌人预设的 SceneController

```
using UnityEngine;
using System.Collections;

public class SceneController : MonoBehaviour {
  [SerializeField] private GameObject enemyPrefab;
  private GameObject _enemy;

  void Update() {

      if (_enemy == null) {
        _enemy = Instantiate(enemyPrefab) as GameObject;
        _enemy.transform.position = new Vector3(0, 1, 0);
        float angle = Random.Range(0, 360);
        _enemy.transform.Rotate(0, angle, 0);
      }
    }
  }
```

序列化变量，用于链接预设对象

一个私有变量，跟踪场景中敌人的实例

这个方法复制了预设对象

只有当场景中没有敌人时才产生一个新敌人

把这个脚本附加到控制器对象上，而在 Inspector 中会显示一个用于设置敌人预设的变量槽。这和公有变量的工作方式类似，但有重要的区别。

警告 建议在 Unity 编辑器中使用带 SerializeField 的私有变量来引用对象，因为想在 Inspector 中显示那个变量，但是不想让其他脚本修改它的值。如第 2 章所述，公有变量默认在 Inspector 中显示(换句话说，它们会被 Unity 序列化)，因此大多数教程和示例代码给所有序列化的值都使用公有变量。但这些变量也因此能被其他脚本修改(毕竟它们是公有变量)；大多数情况下，只在 Inspector 中设置这些值，而不想让其他代码修改它们。

将预设资源从 Project 视图拖到空变量槽；当鼠标靠近时，变量槽会高亮显示，表示哪个对象能链接到这里(见图 3-8)。一旦敌人预设链接到 SceneController 脚本，就可以运行场景，查看代码的行为。像之前一样，房间中心会出现敌人，但如果现在向敌人射击，他将会被新敌人代替。比永远只有一个敌人好多了！

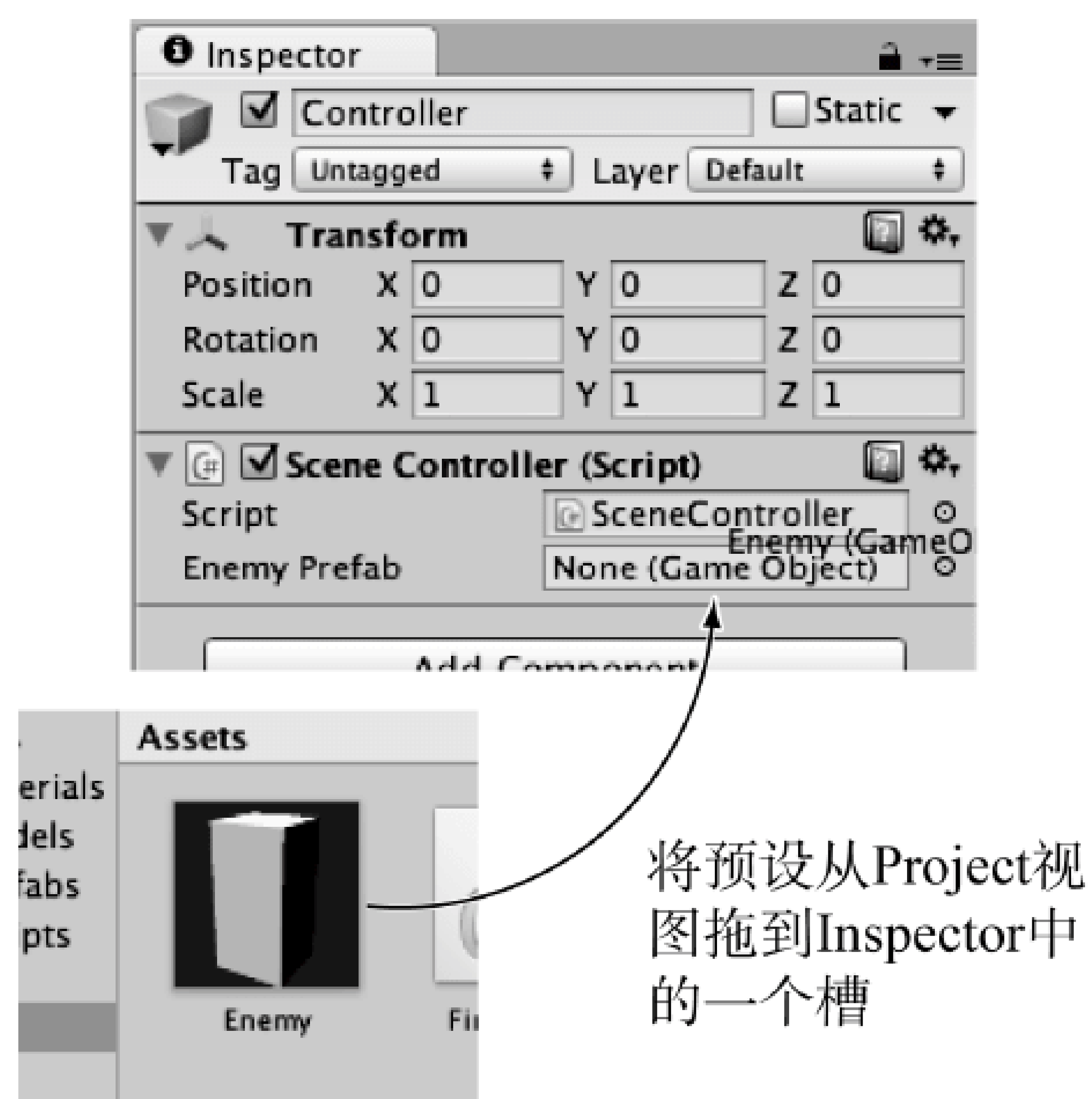


图 3-8　将敌人预设链接到脚本的 Prefab 槽

提示 在众多不同的脚本中，这种将对象拖到 Inspector 变量槽的方法是一种便利的技术。现在将预设链接到脚本，也可以链接到场景中的对象上，甚至链接到指定的组件上(因为代码需要调用那个指定组件上的公有方法)。后面的章节将继续使用这种技术。

这个脚本的核心是 Instantiate()方法，因此注意一下该方法。当实例化预设时，就在场景中创建了一份副本。默认情况下，Instantiate()返回的新对象是通用 Object 类型，但 Object 几乎没什么用，而我们想把它作为 GameObject 处理。在 C#中，使用 as 关键字可以将一种类型的代码对象转换为另一种类型(使用语法 original-object as new-type)。

实例化的对象保存在_enemy 中，它是一个 GameObject 类型的私有变量(要弄清楚预设和预设实例的区别，enemyPrefab 保存预设，而_enemy 保存实例)。if 语句检查保存的对象，确保当_enemy 为空(编码术语为 null)时 Instantiate()只调用一次。_enemy 变量开始为空，因此在开始会话之后，运行一次实例化代码。接着，由 Instantiate()返回的对象保存在_enemy 中，这样不会再次运行实例化代码。

由于敌人被射中时会销毁自己，因此使_enemy 变量变为空，再次运行 Instantiate()。这样，场景中始终有一个敌人。

销毁 GameObject 和内存管理

当对象销毁自身时，已有引用会变为 null，这有点出乎预料。在像 C#一样的内存管理编程语言中，通常不能直接销毁对象，只能解除对它们的引用以便它们能自动销毁。这在 Unity 中依然适用，但 GameObject 是在后台处理的，这让它看起来像是直接销毁的。

为了显示场景中的对象，Unity 需要在场景图中引用所有对象。因此即使移除代码中所有对 GameObject 的引用，它依然会被这个场景图引用，以防止对象被自动销毁。因此，Unity 提供了 Destroy()方法来告诉游戏引擎“将这个对象从场景图中移除”。作为后台功能的一部分，Unity 也重载了==操作符，当检查为 null 时返回 true。技术上，对象依然存在于内存中，但它可能不再存在，因此 Unity 让它显示为 null。调用已销毁对象的 GetInstanceID()方法，就可以确认。

注意，Unity 开发者考虑将这种行为变成更标准的内存管理。如果他们这样做了，那么生成的代码也需要修改，可以通过将(_enemy==null)检查换为一个新参数，例如(_enemy.isDestroyed)来实现。

(如果完全不懂这些讨论，不用担心；这只是不相关的技术讨论，适用于对这些模糊细节感兴趣的人。)

3.5　通过实例化对象进行射击

下面给敌人添加另一个功能。就像给玩家添加功能一样，首先让他们移动——现在，让他们射击！在介绍射线发射时提到，那只是一种实现射击的方法。另一种方法涉及实例化预设，下面通过实例化预设让敌人射回。本节的目标是运行时的结果，如图 3-9 所示。

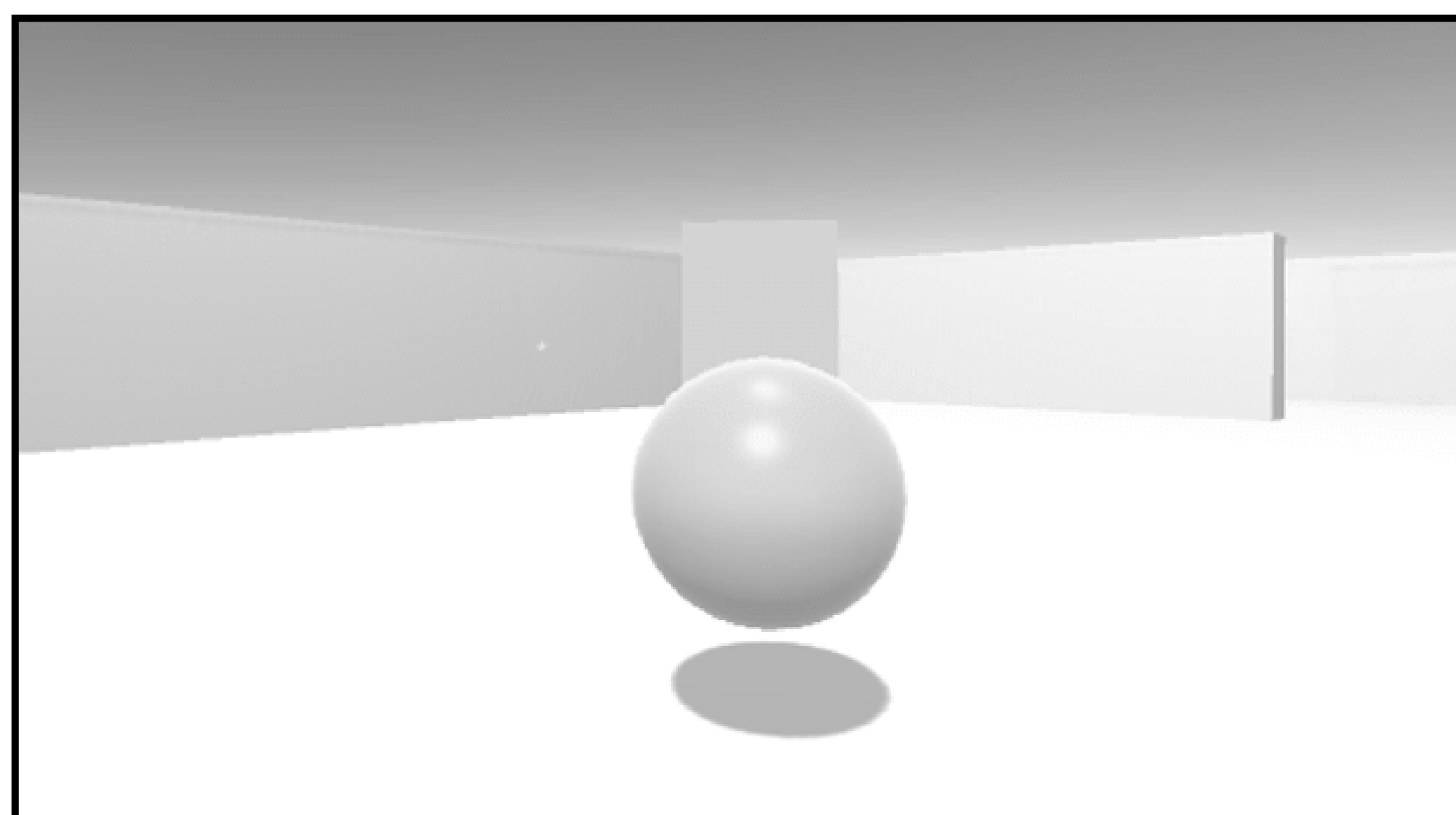


图 3-9　敌人向玩家发射火球

3.5.1　创建子弹预设

接下来的射击将在场景中引入子弹。使用射线发射实现射击基本是瞬间完成的，在鼠标按下时显示击中，但这次敌人发射将从空中飞过的火球。当然，火球移动得非常快，但不是立刻击中，这给玩家一个及时躲避的机会。我们使用碰撞检测而不是射线发射来检测这种碰撞(该碰撞检测系统也防止移动中的玩家穿越墙壁)。

代码将以产生敌人一样的方式产生火球：通过实例化预设。如前面章节所述，创建预设的第一步是在场景中创建一个要成为预设的对象，因此，接下来先创建一个火球，选择 GameObject | 3D Object | Sphere。将新对象重命名为 Fireball。现在创建一个新脚本，也称为 Fireball，并附加到 Fireball 对象上。最后在这个脚本中编写代码，但现在先对 Fireball 对象做一些其他处理。为了让它看起来像个火球而不只是一个灰色的球，给这个对象指定明亮的橙色。类似颜色这样的表面属性由其材质控制。

定义　材质(material)是一组信息，这些信息定义了附加该材质的 3D 对象的表面属性。这些表面属性包括颜色、发光甚至精细的粗糙度。

选择 Assets | Create | Material。命名新材质为 Flame，把它拖放到场景的对象上。在 Project 视图中选择材质，以便在 Inspector 中查看材质的属性。如图 3-10 所示，单击带 Albedo 标签的色卡(这是指向表面主颜色的技术术语)。单击后在它的窗口中会出现颜色拾取器；同时滑动右边的彩虹颜色条和主要选择区，将颜色设置为橙色。

我们还要让材质更明亮，使它看起来更像火焰。调整 Emission 值(Inspector 中的其他属性之一)。其默认值为 0，因此输入 0.3 来提高材质的亮度。

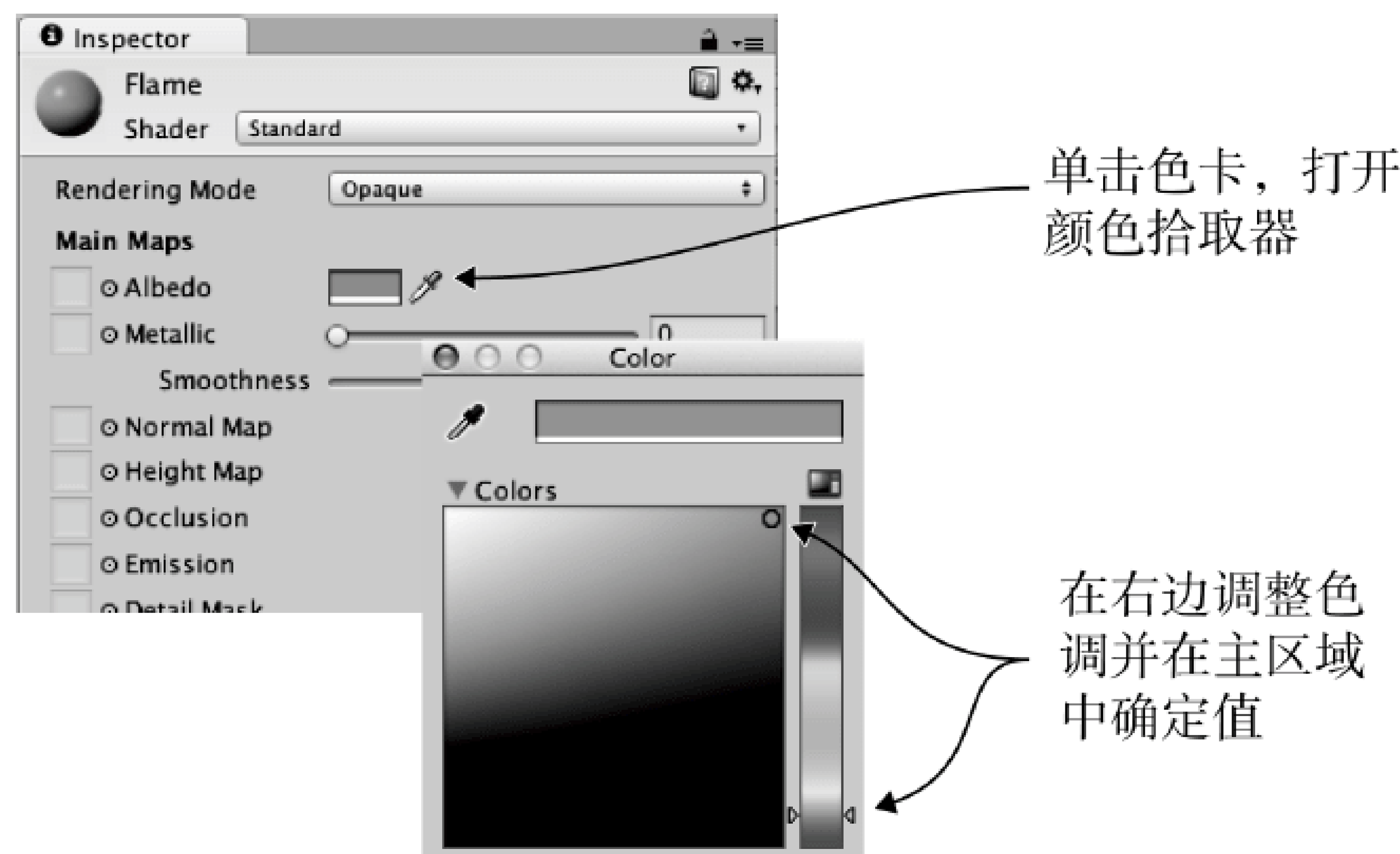


图 3-10　设置材质的颜色

现在把 fireball 对象从 Hierarchy 视图拖到 Project 视图，把该对象变成预设，就像对敌人预设所做的操作那样。现在就有了一个新预设用作子弹！接下来编写代码，来发射那个子弹。

3.5.2　发射子弹并和目标碰撞

为了让敌人发射火球，接下来对它做一些调整。因为识别玩家需要一个新脚本(类似 ReactiveTarget 由代码用于识别目标)，所以先创建新脚本，并命名为 PlayerCharacter。将这个脚本附加到场景中的玩家对象上。

现在打开 WanderingAI 并添加代码清单 3.11 中的代码。

代码清单 3.11　给 WanderingAI 添加发射火球的代码

```
...
[SerizeField] private GameObjectfireballPrefab;
privateGameObject _fireball;
...
if (Physics.SphereCast(ray, 0.75f, out hit)) {
  GameObject hitObject = hit.transform.gameObject;
  if (hitObject.GetComponent<PlayerCharacter>()) {
    if(_fireball == null) {
      _fireball = Instantiate(fireballPrefab) as GameObject;
      _fireball.transform.position =
     transform.TransformPoint(
       Vector3.forward * 1.5f);
      _fireball.transform.rotation = transform.rotation;
    }
  }
  else if (hit.distance< obstacleRange) {
     float angle = Random.Range(-110, 110);
     transform.Rotate(0, angle, 0);
  }
}
...
```

在任何方法前添加这两个字段，就像在 Scene Controller 中那样

使用在 RayShooter 中检查目标对象的方式检查玩家

将火球放在敌人前面并指向同一方向

这里的 Instantiate()方法和 SceneController 中的一样

和在SceneController中检查空GameObject一样的逻辑

注意，代码清单中的所有注释和之前的脚本很像(或者一样)。之前的代码清单展示了发射火球所需的所有代码；现在把代码混在一起，重新组合，使之适应新的环境。就像 SceneController，需要在脚本顶部添加两个 GameObject 字段：一个用于链接预设的序列化变量，一个用于追踪通过代码创建的实例副本的私有变量。在发射射线之后，代码在受击对象上检查 PlayerCharacter；这和射击代码检查受击对象上的 ReactiveTarget 一样。当场景中不存在火球时，代码会实例化一个，就像实例化敌人的代码一样。然而位置和旋转角度不同；这次把实例放到敌人前面，并指向和敌人一样的方向。

一旦所有新代码准备完毕，在 Inspector 中查看组件时，会看到一个新的 Fireball Prefab 槽，与 Enemy Prefab 槽出现在 SceneController 组件中一样。单击 Project 视图中的敌人预设后，Inspector 将显示该对象的组件，就好像在场景中选择对象一样。尽管之前警告过 Unity 编辑预设时提供的界面很粗糙，但如当前所做，这个界面很容易调整对象上的组件。如图 3-11 所示，从 Project 中将火球预设拖动到 Inspector 的 Fireball Prefab 槽上(同样，和之前在 SceneController 上做的一样)。

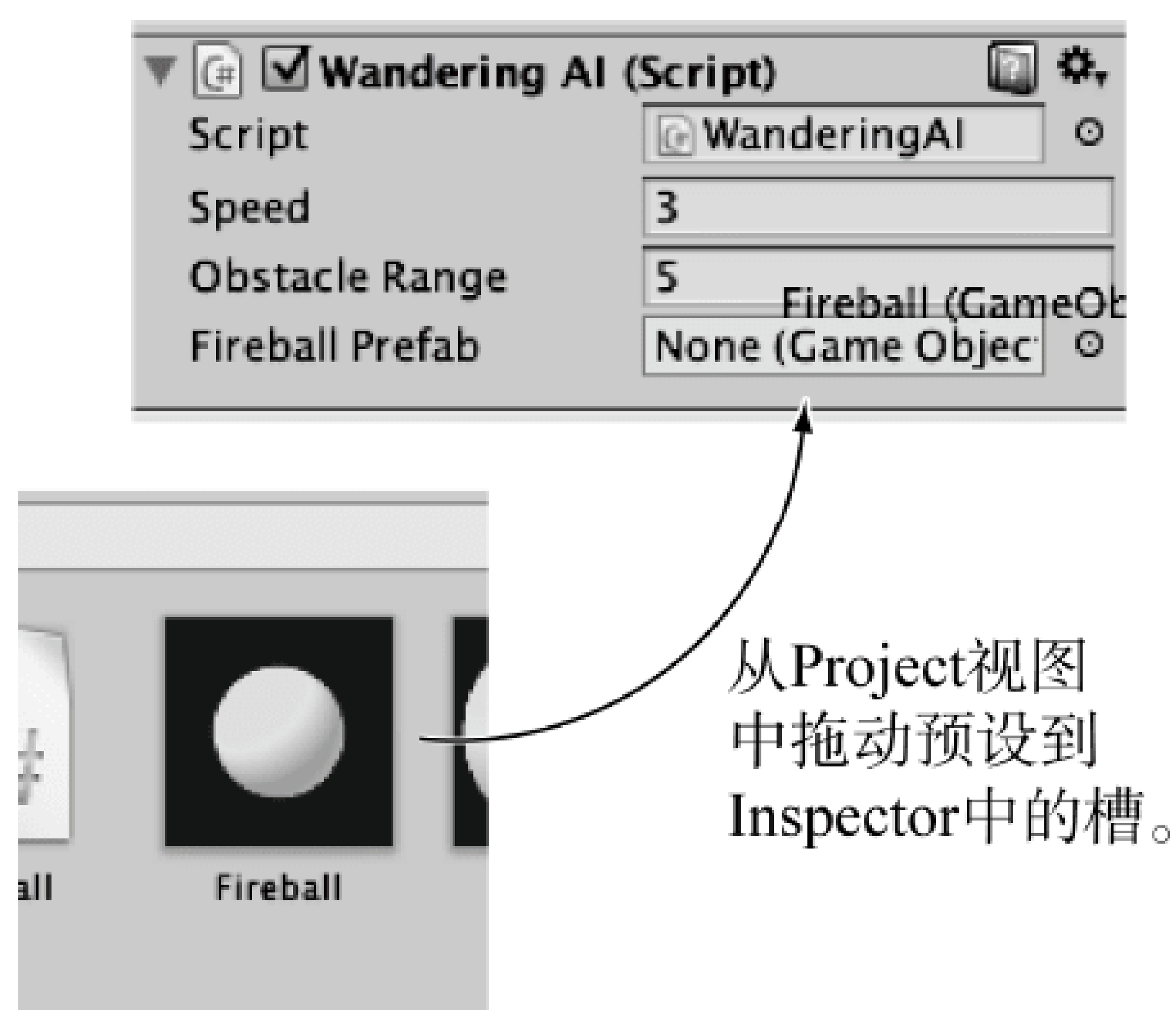


图 3-11　把火球预设链接到脚本的预设槽

现在当玩家在敌人前面时，敌人会向它开火……好，尝试开火。明亮的橙色球体会显示在敌人前面，但它只是停留在那里，因为我们还没对它编写脚本。现在开始编写脚本。代码清单 3.12 展示了 Fireball 脚本的代码。

代码清单 3.12　对碰撞做出反应的 Fireball 脚本

```
using UnityEngine;
using System.Collections;

public class Fireball : MonoBehaviour {
  public float speed = 10.0f;
  public int damage = 1;

  void Update() {
```

```
    transform.Translate(0, 0, speed * Time.deltaTime);
  }

  void OnTriggerEnter(Collider other) {
    PlayerCharacter player = other.GetComponent<PlayerCharacter>();
    if(player != null) {
      Debug.Log("Player hit")
    }
    Destroy(this.gameObject);
  }
}
```

当其他对象和这个触发器碰撞时调用这个方法

检查 other 对象是否为 PlayerCharacter

这段代码最重要的新知识点为 OnTriggerEnter()方法。当对象被碰撞时会自动调用这个方法，诸如和墙壁或玩家碰撞。此时这段代码还不能工作；如果运行它，由于 Update()方法包含了 Translate()代码，火球将向前飞出，但触发器不会执行，在摧毁当前火球之前新的火球会等待。这需要对 Fireball 对象的组件做些调整。第一个修改是让碰撞器成为触发器。为此，选中 Sphere Collider 组件的 Is Trigger 复选框。

提示 将碰撞器组件设置为触发器，依然会对与其他对象接触/重叠做出反应，但它不再阻止其他对象在物理上穿过它。

火球还需要一个 Rigidbody(刚体)，这个组件在 Unity 中由物理系统使用。给火球添加 Rigidbody 组件，确保物理系统能为对象注册碰撞触发器。在 Inspector 中，单击 Add Component 按钮，选择 Physics | Rigidbody。在添加的组件中，取消选中 Use Gravity(见图 3-12)，以便火球不会因为重力而下落。

现在运行代码，当火球击中某物时将被摧毁。因为只要场景中不存在火球，就运行火球生成代码，所以敌人向玩家发射更多的火球。现在对于向玩家发射只剩一件事情要做：让玩家对受击做出反应。

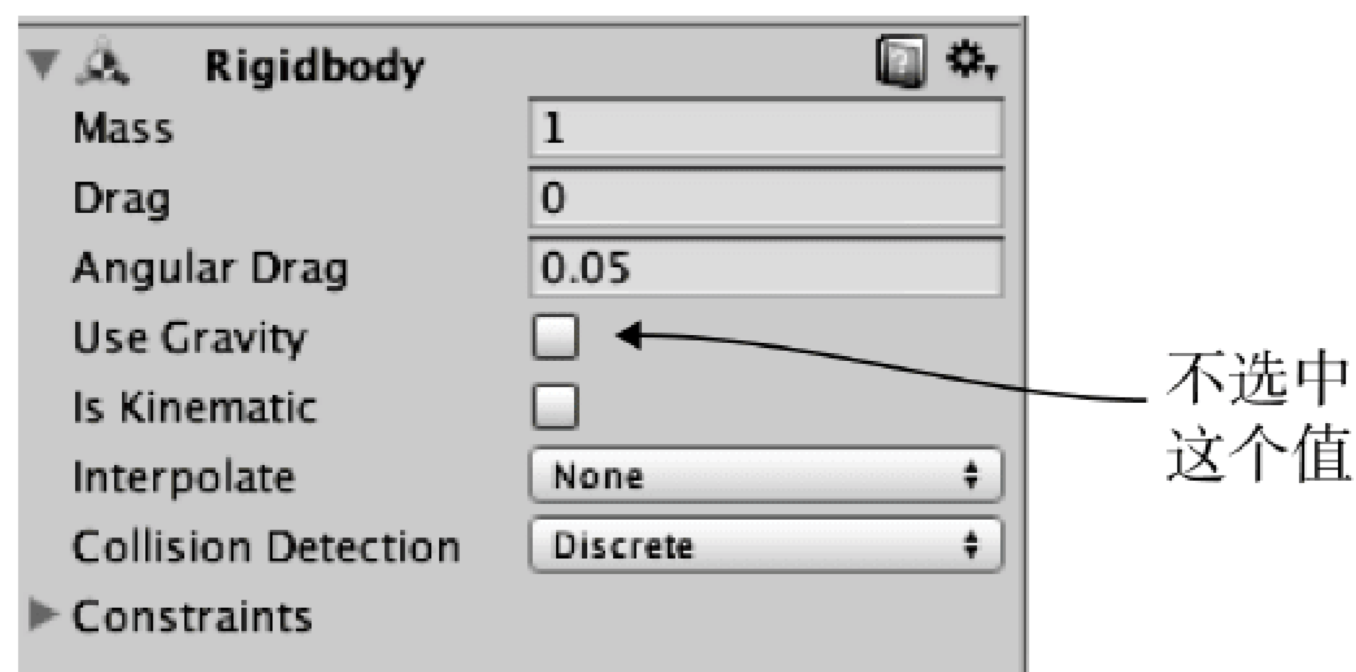


图 3-12　关闭 Rigidbody 组件的重力

3.5.3　伤害玩家

之前创建了 PlayerCharacter 脚本，但它还是空的。现在编写代码，让它对受击做

出反应，如代码清单 3.13 所示。

代码清单 3.13　能受到伤害的玩家

```
using UnityEngine;
using System.Collections;

public class PlayerCharacter :MonoBehaviour {
  private int _health;

  void Start() {
  _health = 5;           ← 初始化血量
  }

  public void Hurt(int damage) {
    _health -= damage;                  ← 减少玩家的血量
    Debug.Log("Health: " + _health);
  }
}
```

代码清单中定义的一个字段用于存储玩家的血量，并且在命令中减少血量。后续章节将重温文本显示，以在屏幕上展示信息，但现在，只需要使用调试消息显示关于玩家的血量信息即可。

现在需要返回 Fireball 脚本，调用玩家的 Hurt()方法。将 Fireball 脚本中的 debug 那一行替换成 player.Hurt(damage)，以通知玩家他们被击中。而这是本章最后需要的代码！

本章的内容很多，介绍了很多代码。通过第 2 章和本章，第一人称射击游戏中现在已经具备了大多数功能。

3.6　小结

- 射线是发射到场景中的虚拟线。
- 对于射击和感知到的障碍物，用射线做一个射线投射。
- 通过基础 AI 让角色四处行走。
- 通过实例化预设产生新对象。
- 协程用于随着时间的推移逐步执行函数。

第 4 章

为游戏开发图形

本章涵盖：

- 了解美术资源
- 了解白盒
- 在 Unity 中使用 2D 图像
- 导入自定义 3D 模型
- 构造粒子效果

之前主要关注游戏的功能，没有太多考虑游戏的外观。这并非偶然——本书主要介绍如何在 Unity 中编写游戏。然而，了解如何提升视觉效果也很重要。在回到本书关注游戏中不同部分的编程之前，先用一章的篇幅学习游戏美术，这样项目不会在收尾时仍然到处是空盒子。

游戏中的所有可视化内容由美术资源(art asset)组成。但美术资源具体是指什么呢？

4.1 了解美术资源

美术资源是由游戏使用的可视化信息(通常是文件)的独立单元。它是所有可视化内容的涵盖性术语；图像文件是美术资源，3D 模型也是美术资源。实际上，术语“美术资源”仅仅是资源的一个特例，之前学习的用于游戏的任何文件都是资源(例如脚本)——因此它们位于 Unity 的主 Assets 文件夹中。表 4-1 列出并描述了用于构建游戏的

五种主要类型的美术资源。

表 4-1　美术资源的类型

美术资源的类型	该类型的定义
2D 图像	扁平的图片。以真实世界作类比，2D 图像就像图画和图片
3D 模型	3D 虚拟对象(通常是“网格对象”的同义词)。以真实世界作类比，3D 模型就像雕塑
材质	该组信息定义了附加材质的对象的表面属性。这些表面属性包括颜色、发光等
动画	动画打包了关联对象的运动信息。这些信息描述提前创建的运动序列，而不是及时计算位置的代码
粒子系统	一个用于创建并控制大量小型对象的规则机制。很多可视化效果通过这种方式实现，例如火焰、烟雾或喷水

为新游戏创建美术资源通常从 2D 图像或 3D 模型开始，因为它们是其他所有资源依赖的基础资源。顾名思义，2D 图像是 2D 图形的基础，而 3D 模型是 3D 图形的基础。具体而言，2D 图像是扁平的图片；即使之前不熟悉游戏美术，也已经熟悉了网站上用于图形的 2D 图像。对于 3D 模型，新手可能不熟悉，因此下面介绍它的定义。

定义　模型是 3D 虚拟对象。第 1 章介绍了网格对象这一术语；3D 模型实际上是其同义词。这两个术语通常可以互换，但网格对象严格上指的是 3D 对象(连线和形状)的几何结构，而模型更有歧义，它通常包括对象的其他属性。

列表中接下来的两种资源类型是材质和动画。不像 2D 图像和 3D 模型，材质和动画单独使用时没有任何作用，新手很难理解。2D 图像和 3D 模型通过和真实世界对比则很容易理解：前者是图画，后者是雕塑。材质和动画不能直接关联到真实世界的例子。实际上，它们都是 3D 模型上层抽象信息的打包。例如，在第 3 章介绍基础知识时就已介绍了材质。

定义　材质是一组信息，它定义了附加材质的对象的表面属性(颜色、发光等)。分别定义表面属性，多个对象就可以共享一个材质(例如，所有的城堡墙)。

下面继续对美术资源进行类比，可以将材质想象为构成雕塑的媒介(泥土、黄铜、大理石等)。类似地，动画是附加到可视化对象上的抽象层信息。

定义　动画是定义关联对象的运动信息的信息包。因为这些运动能独立于对象自身定义，所以它们能以混合-匹配的方式用于多个对象上。

举个具体的例子，思考一下四处游走的角色。角色的位置通过游戏代码来处理(例如第 2 章编写的移动脚本)。但脚踩踏地板，挥动手臂，扭动臀部这些具体的运动便是

回放的动画序列，动画序列就是美术资源。

为了帮助理解动画和 3D 模型是如何关联在一起的，接下来类比操纵木偶：3D 模型是木偶，动画器(Animator)是让木偶移动的操纵者，而动画则是木偶运动的记录。以这种方式定义的运动是提前创建好的，它通常进行少量运动，并且不改变对象的位置。这和前面章节中大量运动的代码形成对比。表 4-1 中的最后一种美术资源为粒子系统。

定义　粒子系统是用于生成和控制大量运动对象的规则机制。这些运动对象通常较小——因此称为粒子，但它们不一定非要很小。

粒子系统用于创建可视化效果，例如火焰、烟雾或喷水。粒子(也就是粒子系统控制下的独立对象)可以是任何网格对象，但对于大多数效果，粒子是一个显示图片(例如火星或扩散的烟雾)的方块。

创建游戏美术资源的许多工作是在外部软件中完成，而不是在 Unity 中完成。材质和粒子系统在 Unity 中创建，但其他美术资源使用外部软件创建。参考附录 B 可以了解更多相关的外部软件；有很多美术应用程序用于创建 3D 模型和动画。在外部工具中创建的 3D 模型最终保存为美术资源，并导入到 Unity 中。在附录 C 中使用 Blender (可以从 www.blender.org 下载它)讲解如何建模，但这仅仅是因为 Blender 是开源的，它可以用于所有读者。

注意　本章下载的项目包含一个名为“scratch”的文件夹。“scratch”文件夹和 Unity 项目在同一个位置，但它不是 Unity 项目的一部分；“scratch”中放置的是额外的外部文件。

在完成本章项目的过程中，将学习大部分类型的美术资源示例(现在讲解动画还有一点复杂，因此在本书后面提及)。我们将使用 2D 图像、3D 模型、材质和粒子系统构建一个场景。一些示例将使用已有的美术资源并学习如何将它们导入到 Unity 中，但一些示例(特别是使用粒子系统时)将在 Unity 中从头创建美术资源。

本章仅触及游戏美术资源创建的皮毛。本书主要介绍在 Unity 中如何编程，大量涉及美术学科的内容将减少编程方面的内容。创建游戏美术资源本身就是一个巨大的话题，其内容能轻易填满几本书。大多数情况下，游戏编程人员需要一个精通美术学科的搭档。也就是说，游戏编程人员了解 Unity 如何处理美术资源，甚至自己创建粗糙的替代资源(常称为程序员美术)是极其有用的。

注意　本章不需要利用前面章节中的项目。但要有像第 2 章那样的移动脚本，才能在要构建的场景中走动；如果需要，可以从下载的项目中获取玩家对象和脚本。类似地，本章最终会有移动的对象，它与第 3 章创建的移动对象非常类似。

4.2　构建基础 3D 场景：白盒

第一个要创建的内容是白盒。这个过程通常是在计算机上创建关卡的第一步(在纸上设计关卡之后)。顾名思义，使用空白几何体(即白盒)勾勒出场景的墙壁。查看不同美术资源的列表，就会发现这个空白场景是 3D 模型最基础的一类，它提供了显示 2D 图像的基础。如果回想一下第 2 章创建的基础场景，就会发现整个场景都是基础的白盒(只是尚未学习这个术语)。本节将重做第 2 章开头的一些工作，但这次会快速完成这个过程，然后更多地讨论新的术语。

注意　另一个常用的术语是灰盒。其含义其实是相同的。这里使用白盒，是因为它是我先学到的术语，使用灰盒也是可以接受的。实际使用的颜色和名称有所不同，就像蓝图不一定是蓝色的。

4.2.1　白盒的解释

使用空白几何体描绘场景草图有一些目标。首先，该过程允许快速构建“草稿”，日后再慢慢完善它。该行为同关卡设计和/或关卡设计师密切相关。

定义　关卡设计是在游戏中规划和创建场景的过程。关卡设计师从事关卡的设计工作。

随着游戏开发团队的发展壮大，团队成员越来越专业，通用的关卡创建工作流是关卡设计师通过白盒创建第一个版本的关卡。接着，这个粗糙的关卡提交给美术团队，打磨视觉效果。即使团队很小，由同一个人负责设计关卡和创建游戏的美术资源，先创建白盒、再打磨视觉效果的工作流程也非常不错。毕竟游戏需要从一个地方开始，而白盒为构建视觉效果提供了清晰的基础。

使用白盒的第二个目标是关卡能快速达到可玩状态。关卡可能还没完成(实际上，刚刚创建好白盒的关卡距离完成还有很大的差距)，但这个粗糙的版本可以运行，能够玩游戏。至少，玩家可以在场景中走动(参考第 2 章的演示游戏)。注意，现在就可以进行测试，确保关卡能正常运行(例如房间大小对于这个游戏是否合适)，以后再投入更多的时间和精力处理细节工作。在白盒布景中时，如果出错(例如空间需要更大一些)，也很容易修改并重新测试。

而且，能在构建中的关卡中玩游戏有利于提升士气。不要小觑这种好处：为场景构造所有视觉效果需要大量时间，如果在体验游戏的任何方面之前需要等待很长的时间，会让人觉得进度缓慢。白盒能立刻构建完整(但非常基本)的关卡，在逐步改善游戏的同时能够玩游戏，这将激动人心。

明白了为什么关卡从白盒开始，下面开始构建关卡！

4.2.2　为关卡绘制地板平面图

可以根据纸上设计的关卡在计算机中构建关卡。这里不准备大篇幅讨论关卡的设计。第 2 章提到游戏的设计，而关卡设计(它是游戏设计的子集)是一个很大的分支，它本身的内容足够填满一整本书。为了便于讨论，下面绘制一个基本的关卡，对平面图做点小设计从而设定要达到的目标。

图 4-1 是一个简单布局的顶视图，其中的四个房间由中心走廊连接起来。目前需要的平面图如下：通过内墙分开的一些区域。在真正的游戏中，平面图会更复杂，可能包括敌人和物品。

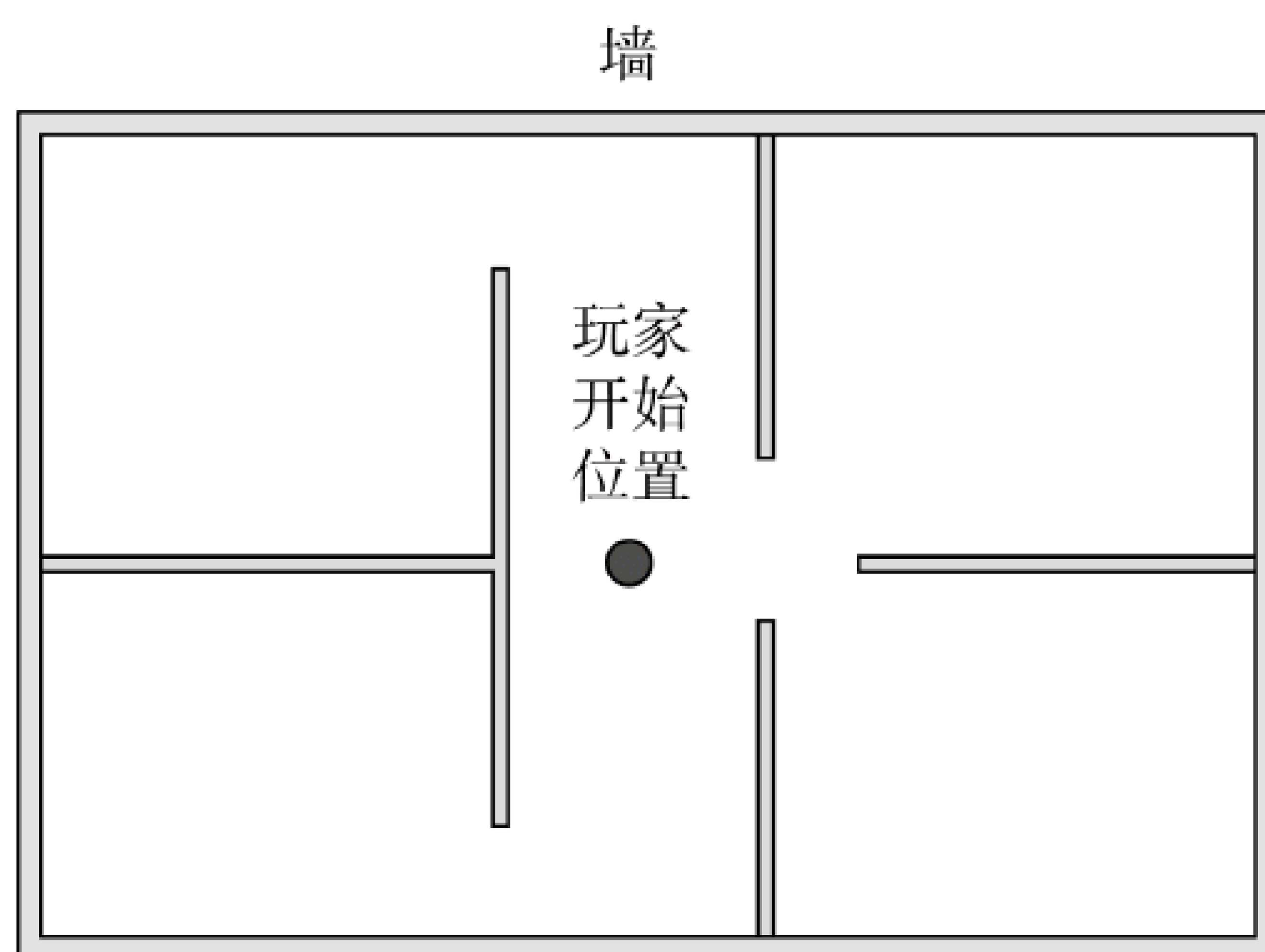


图 4-1　关卡的地板平面图：四个房间和一条中心走廊

可以通过创建这个地板平面图来练习白盒，也可以绘制简单的关卡来练习这个步骤。房间的具体布局在这个练习中并不重要。重要的是绘制好一张地板平面图，这样才能继续下一步。

4.2.3　根据平面图布局几何体

根据绘制好的地板平面图构建白盒场景，需要定位并缩放一系列空盒子，使之成为图中的墙壁。如第 2 章的 2.2.1 节所述，选择 GameObject | 3D Object | Cube，创建空盒子，然后根据需要定位和缩放它。

注意　这个步骤不是必需的，可以使用下载项目中的 QuadsBox 对象来替代立方体对象。这个对象是由 6 个独立的块组成的立方体，所以对它应用材质时更灵活。是否使用这个对象取决于工作流的需求；例如，这里不使用 QuadsBox，因为所有白盒几何体在以后都会被新的美术资源替换。

使用 CSG 编辑关卡

在本章介绍的工作流中，首先用基本几何体构建关卡，然后在外部 3D 美术工具中构建最终的关卡几何体。另一种编辑关卡的方法称为 CSG(Constructive Solid Geometry，构造立体几何)。在这种方法中，不使用 Unity 的基本几何体，而是使用称为“画笔”的形状，从最初的原型到最终的几何体都是在 Unity 中构建的。

Unity 有多个 CSG 插件。一种选择是 SabreCSG，它是一套开源的 CSG 工具。更多信息请访问 sabrecsg.com。

第一个对象是场景的地板；在 Inspector 中，重命名对象，把它的 Y 轴调低为−0.5，以考虑盒子自身的高度(如图 4-2 所示)。接着沿着 X 轴和 Z 轴缩放该对象。



图 4-2　为制作地板修改位置和缩放盒子的 Inspector 视图

重复这些步骤，创建场景中的墙壁。把墙壁作为一个公共基础对象的子对象，可以让 Hierarchy 视图更干净(记住使根对象位于(0, 0, 0)，然后在 Hierarchy 中把墙壁拖到根对象上)，但这不是必需的。接着也将一些简单的光源放到场景中，以便能看清场景中的对象。回顾第 2 章的内容可知，选择 GameObject 菜单下的 Light 子菜单创建灯光。一旦完成了白盒的工作，关卡应该如图 4-3 所示。

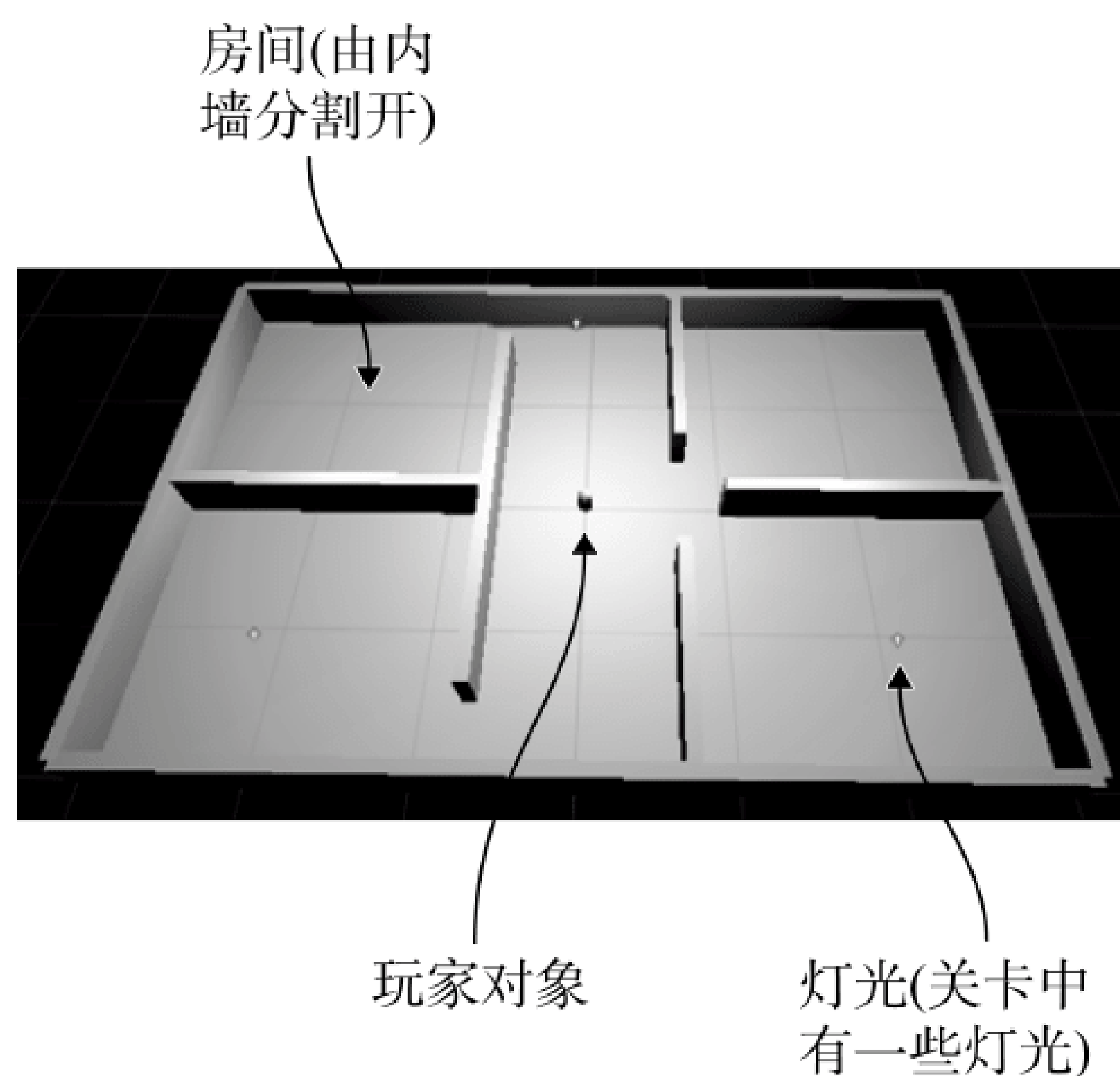


图 4-3　图 4-1 中地板平面图的白盒关卡

设置玩家对象或摄像机来移动(通过角色控制器和移动脚本创建玩家，如果需要完整的解释，请参阅第 2 章的相关内容)。现在就能在基本场景中移动，体验并测试前面完成的工作。而这正是使用白盒的方式！很简单——但现在只有空白的几何体，接下来在墙壁上使用图片点缀几何体。

为外部美术工具导出白盒几何体

为关卡打磨视觉效果的许多工作是在诸如 Blender 这样的外部 3D 美术应用中完成的。因此，可能需要在美术工具中引用白盒几何体。默认情况下，在 Unity 内部创建的几何体没有导出选项。但第三方脚本可以为编辑器添加这个功能。大多数这种脚本允许你选择场景中的几何体并单击 Export 按钮。

这些自定义脚本通常将几何体作为 OBJ 文件导出(OBJ 是本章后面将讨论的几种文件类型之一)。在 Unity3D 网站上，单击搜索按钮并输入 obj exporter。或者可以从如下网址获得一个示例：http://wiki.unity3d.com/index.php?title=ObjExporter。

4.3　使用 2D 图像给场景贴图

当前的关卡是一个粗略的草稿。它可以进行游戏，但很明显，在场景的视觉体验上还有很多工作要做。接下来提高关卡外观的步骤是应用贴图。

定义　贴图是用于提高 3D 图形效果的 2D 图像。这是贴图这个术语完整的解释，只要认为对贴图任何不同形式的使用都是这个术语定义的一部分，就不会混淆该定义。不管图像如何使用，它仍然是贴图。

注意　贴图一词通常用作动词和名词。除了名词定义以外，这个词语还表示在 3D 图形中使用 2D 图像的这种行为。

贴图在 3D 图形中有几个作用，最直接的作用是显示在 3D 模型的表面上。本章后面将讨论如何在复杂的模型上显示贴图，但对于白盒关卡，2D 图像就像墙纸覆盖在墙壁上一样(如图 4-4 所示)。

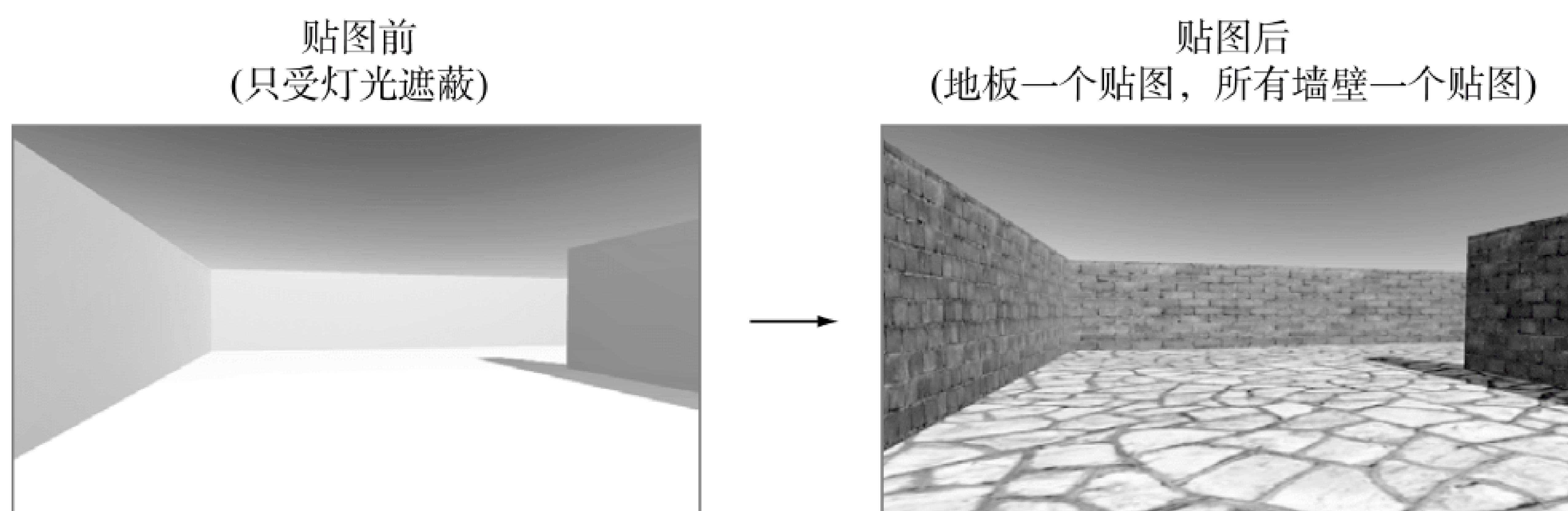


图 4-4　关卡贴图前后的对比

从图 4-4 中的贴图对比可以发现，贴图把明显不真实的数字建筑变成砖墙。贴图的其他用途包括用于裁剪形状，法线贴图使表面产生凹凸。附录 D 提及的资源中提供了贴图的更多信息。

4.3.1　选择文件格式

2D 图像可以保存为不同的文件格式，应该使用哪一种格式呢？Unity 支持多种不同的文件格式，可以选择表 4-2 中的任何一种。

表 4-2　Unity 支持的 2D 图像文件格式

文 件 类 型	优　缺　点
PNG	通常用于万维网。无损压缩，带透明通道
JPG	通常用于万维网。有损压缩，无透明通道

(续表)

文件类型	优缺点
GIF	通常用于万维网。有损压缩，无透明通道(技术上来讲，损耗并不是压缩造成的，而是当图片转为八位时导致数据丢失。最终导致了损耗)
BMP	Windows 上默认的图像格式。无压缩，无透明通道
TGA	通常用于 3D 图形。其他地方不常用。无损压缩或不压缩，带透明通道
TIFF	通常用于数字相片和出版。无损压缩或不压缩，无透明通道
PICT	旧 Macs 系统上的默认图像格式。有损压缩，无透明通道
PSD	Photoshop 原生文件格式。无压缩，有透明通道。使用这种文件格式的主要原因是可以直接使用 Photoshop 文件

定义 透明通道用于保存图像中的透明信息。可见颜色来自三个通道的信息：红、绿、蓝。Alpha 是附加的通道信息，它是不可见的，但控制了图像的可见性。

尽管 Unity 能导入表 4-2 中的任意图像，并用作贴图，但不同文件格式支持的特性有很大区别。对于作为贴图导入的 2D 图像，两个因素特别重要：图像的压缩方式，以及是否有透明通道。透明通道是一个直接的考虑因素：因为透明通道在 3D 图形中经常使用，图像有透明通道会更好。图像压缩是一个比较复杂的考虑因素：可以归结为“有损压缩不好”：不压缩和无损压缩能够保证图像的品质，而有损压缩降低了图像的品质(因此称为有损)，从而减小了文件的大小。

由于上述两个考虑因素，推荐两种文件格式作为 Unity 贴图，分别是 PNG 和 TGA。在 PNG 广泛应用于互联网之前，TGA 曾是 3D 图形中最受欢迎的贴图文件格式；如今 PNG 在技术上几乎可以和 TGA 相媲美，但使用更广，因为 PNG 可用于网络和贴图。PSD 通常也推荐为 Unity 贴图格式，因为它是一种高级文件格式，便于将 Photoshop 中处理的文件直接用于 Unity。但最好工作文件和导出到 Unity 的已完成文件分离开来(这和接下来介绍的 3D 模型文件是同一个道理)。

结论是，为示例项目提供的所有图像都是 PNG 文件格式，建议读者也使用这种文件格式。下面将一些图片加入到 Unity 中，并把它们应用到空白场景。

4.3.2　导入图像文件

接下来创建/准备要使用的贴图。用于给关卡贴图的图像通常是可平铺的，因此它们能在地板之类的大平面上重复平铺。

定义 可平铺图像(有时称为无缝平铺图像)是排列在一起时对边能相互匹配的图像。这种图片能重复平铺，没有任何可见的缝隙。3D 贴图的概念就像网页上的壁纸一样。

可以通过几种不同的方式获得可平铺图像，例如处理照片，甚至手绘它们。可以在很多网站和书籍找到这些技术的教程和说明，但在此不深入介绍该项技术。从一些为3D艺术家提供这种图像的网站中获取一些可平铺图像。例如，从网站www.textures.com/(如图 4-5 所示)中获取一些图像，用于关卡中的墙壁和地板。可以寻找一些适合地板和墙壁的图像。

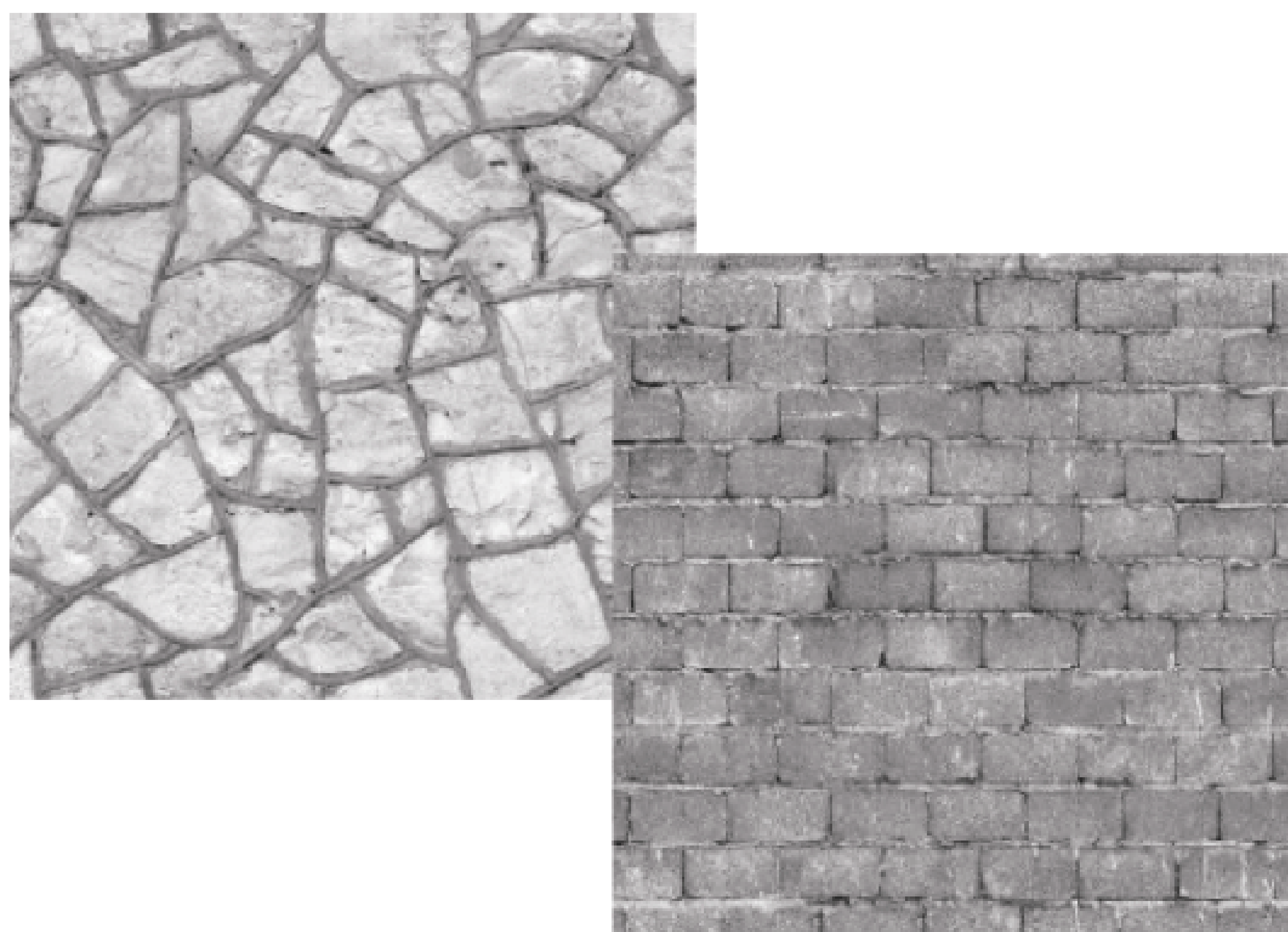


图 4-5　从 textures.com 获取的石头和砖块的无缝贴图

下载需要的图像，准备将它们用作贴图。从技术上讲，可以在下载这些图像之后直接使用它们，但这些图像用作贴图并不完美。尽管它们的确是可平铺的(使用这些图像的一个重要原因)，但其大小并不合适，其文件格式也不正确。贴图大小应该为 2 的幂。出于技术上的有效性，显卡希望处理的贴图大小是 2 的 N 次幂：4、8、16、32、64、128、256、512、1024、2048(下个数字为 4096，但目前该尺寸的图像不适合用作贴图)。在图像编辑器中(Photoshop、GIMP或其他软件，参考附录 B)，把下载的图像缩放为 256×256，然后保存为 PNG 格式。

现在，在计算机中将文件从原来的位置拖到 Unity 的 Project 视图中。这会把该文件复制到 Unity 项目中(如图 4-6 所示)，此时将它们导入为贴图，能用于 3D 场景中。如果觉得拖动文件比较别扭，也可以在 Project 视图中右击并选择 Import New Asset，打开文件拾取器。

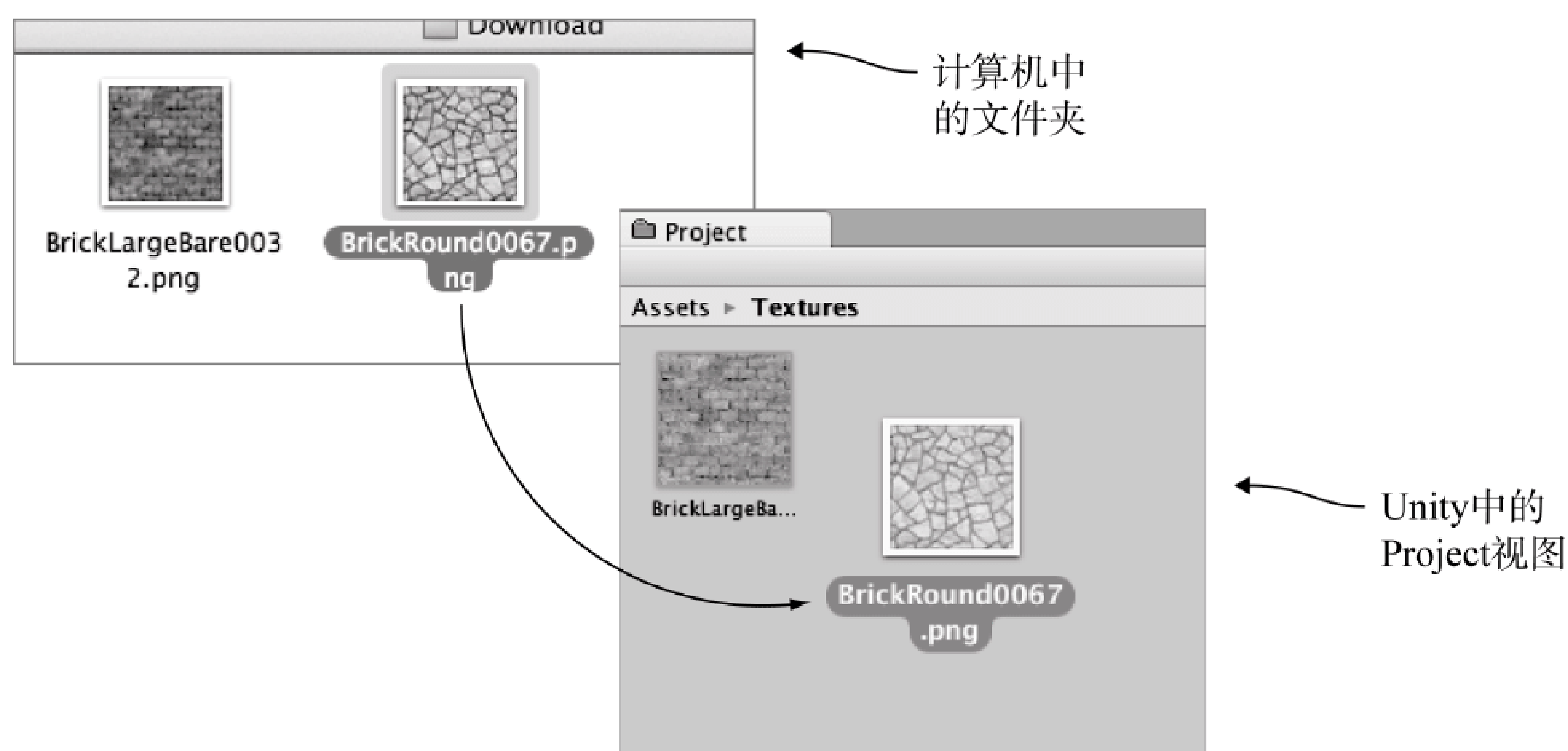


图 4-6　把 Unity 外的图像拖到 Project 视图来导入它们

定义　随着项目变得越来越复杂，最好将资源组织到不同的目录下。在 Project 视图中，创建 Scripts 和 Textures 文件夹并将资源移动到相应的文件夹。只需要简单拖动文件到新文件夹即可。

警告 Unity 中有一些关键字也用作文件夹的名称，这些特殊文件夹中的文件会以特殊的方式处理。这些关键字是 Resources、Plugins、Editor 和 Gizmos。后面将介绍这些特殊文件夹的作用，但现在只需要避免将文件夹命名为这些关键字即可。

现在图像已经作为贴图导入到 Unity 中，可以使用了。但如何将贴图应用到场景的对象上呢？

4.3.3 应用图像

从技术上讲，贴图不能直接应用到几何体上，但贴图可以是材质的一部分，而材质可以应用到几何体上。材质是定义表面属性的一系列信息；该信息可以包括显示在表面上的贴图。这种间接应用方式很重要，因为同一贴图可以用于多个材质。也就是说，通常每个贴图用于一种不同的材质，因此为了方便，Unity 允许将一个贴图拖到对象上并自动创建一个新材质。如果从 Project 视图中将一个贴图拖到场景的对象上，Unity 将创建新的材质，并将这个材质应用到对象上。图 4-7 演示了这种操作。现在尝试将贴图应用到地板上。

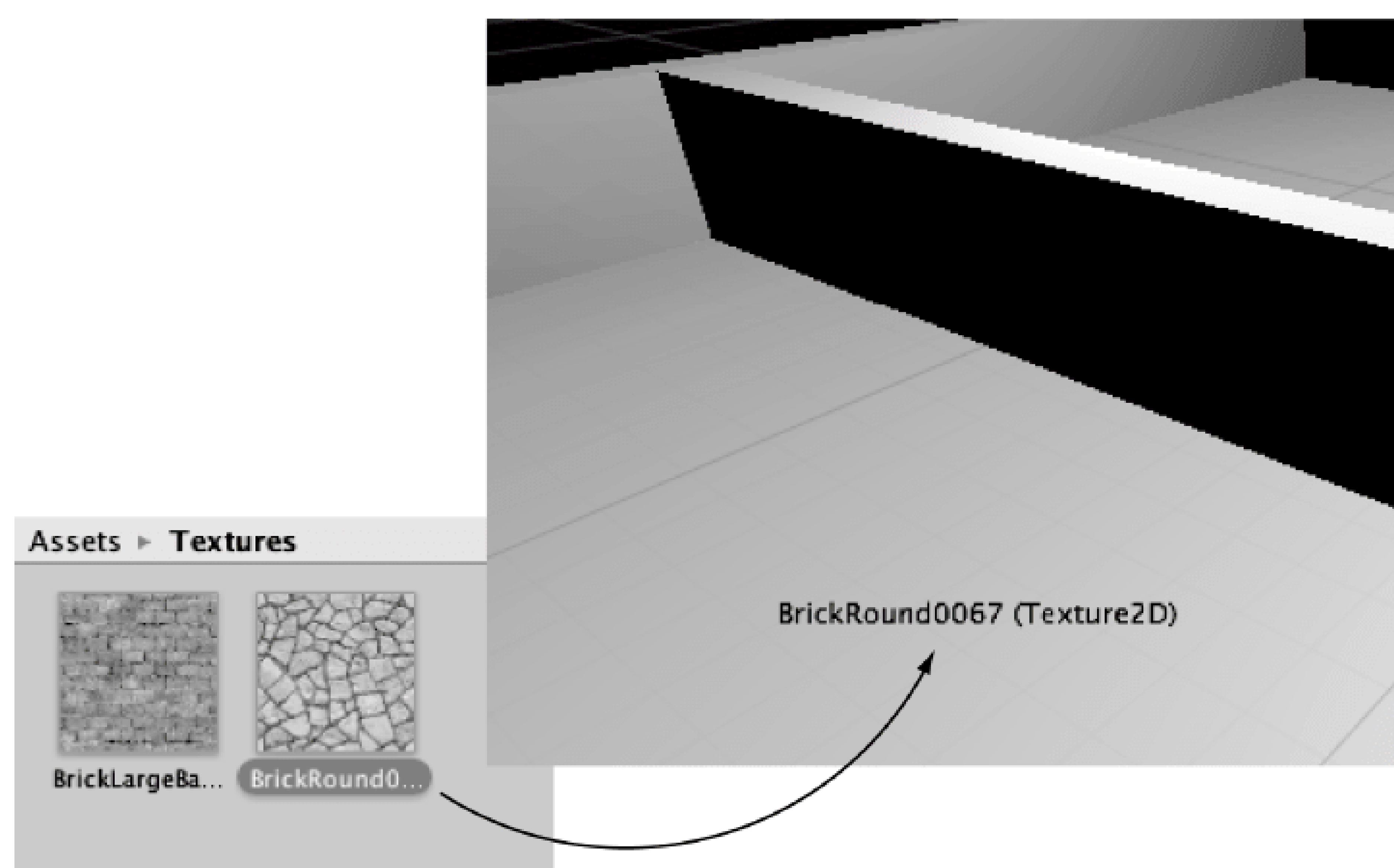


图 4-7 一种应用贴图的方式是把贴图从 Project 拖到场景对象上

除了上述自动创建材质的便捷方式外，创建材质的“正常”方式是使用 Assets 菜单的 Create 子菜单，新资源显示在 Project 视图中。现在选择材质，使它的属性显示在 Inspector 中(如图 4-8 所示)，将贴图拖到主贴图槽，该设置称为 Albedo(这是基础色的技术术语)，而贴图槽是面板边上的方块。此时，从 Project 将材质拖动到场景中的对象上，将材质应用到那个对象。现在尝试这些步骤，给墙壁贴图：创建一个新材质，将墙壁贴图拖动到这个材质上，接着将材质拖动到场景中的墙上。

现在，石头和砖块图像应该出现在地板和墙壁对象的表面上，但图像现在看起来被拉伸了并且相当模糊。这是因为拉伸了这个图像，以覆盖整个地板。下面需要设置图像在地板表面重复平铺的次数。为此可以使用材质的平铺属性来设置；在 Project 中选择

材质，接着在 Inspector 中修改平铺数目(为 X 和 Y 值设置每个方向平铺的次数)。

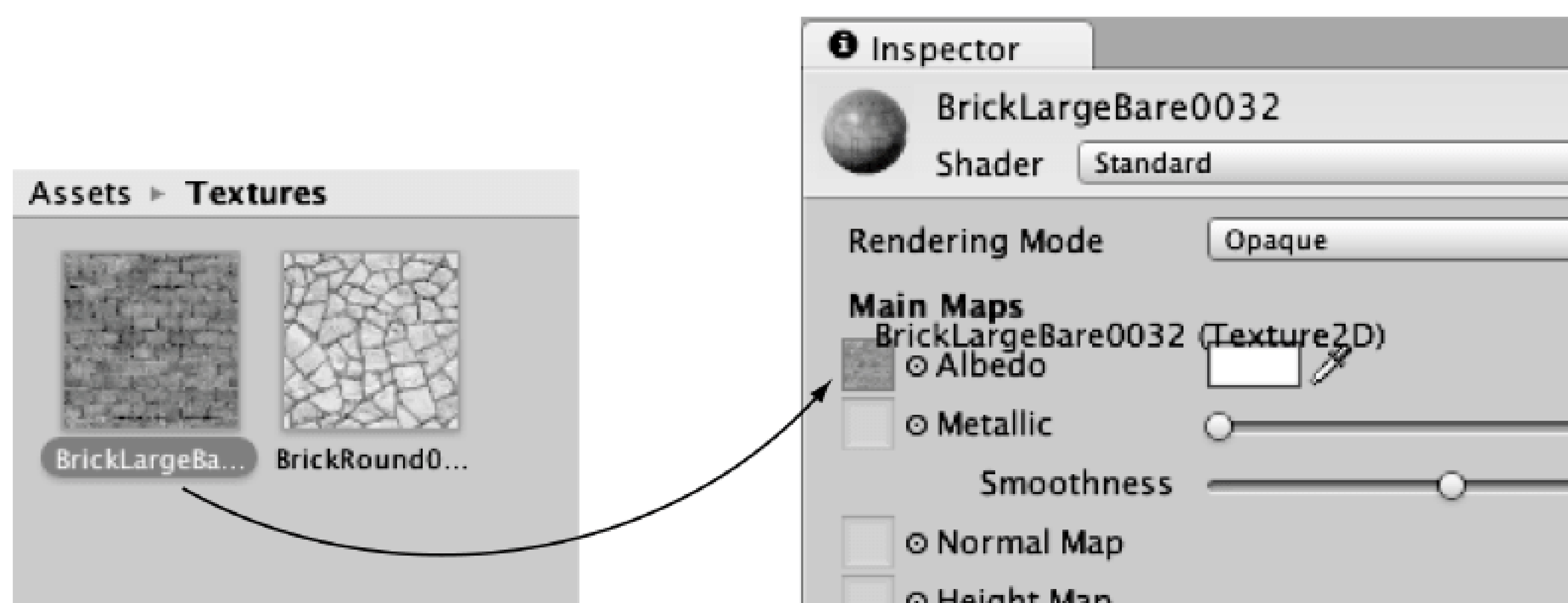


图 4-8　选择材质，在 Inspector 中观察它，接着将贴图拖动到材质属性

确保设置的是 Main Maps 而不是 Secondary Maps(这个材质支持次级贴图，得到高级效果)。默认平铺为 1(即不平铺，直接拉伸图像，以覆盖整个表面)。修改平铺数目为 8，观察场景中发生了什么。改变两个材质的平铺数目，使它们看起来更美观，

注意 像这样调整平铺属性，只适用于白盒几何体的贴图；在打磨好的游戏中，地板和墙壁用更复杂的美术工具构建，包括设置它们的贴图。

现在已经将贴图应用到场景的地板和墙壁上！还可以将贴图应用到场景的天空，接下来介绍这个过程。

4.4　使用贴图图像产生天空视觉效果

砖块和石头贴图让墙壁和地板看起来更自然。而当前天空依然是空的、显得不真实。我们也想让天空看起来更真实。为此，最常见的做法是使用天空图片进行特殊的贴图。

4.4.1　什么是天空盒

默认情况下，摄像机的背景色为深蓝色。通常这个默认颜色填充了视图中任何空白的区域(例如，在场景墙壁之上的区域)，但可以渲染天空图片，作为背景。此时需要使用天空盒的概念。

定义 天空盒是一个包围摄像机的立方体，这个立方体的每个面都用天空图片贴图。不管摄像机面向什么方向，它看到的都是天空图片。

正确实现天空盒会很棘手。图 4-9 是天空盒工作原理的图解。需要一些渲染技巧使天空盒显示为远处的背景。幸运的是，Unity 已经处理好了这些。

新场景实际上已经自带了很简单的天空盒。这正是天空从明亮渐变到暗蓝，而不是一种暗蓝色的原因。如果打开光源窗口(Window | Lighting | Settings)，会看到第一个

设置是 Skybox Material，而这个设置的槽显示为 Default。这个设置位于 Environment Lighting 面板中，这个窗口中包含许多与 Unity 中高级光源系统关联的设置面板，但现在只需要考虑第一个设置。

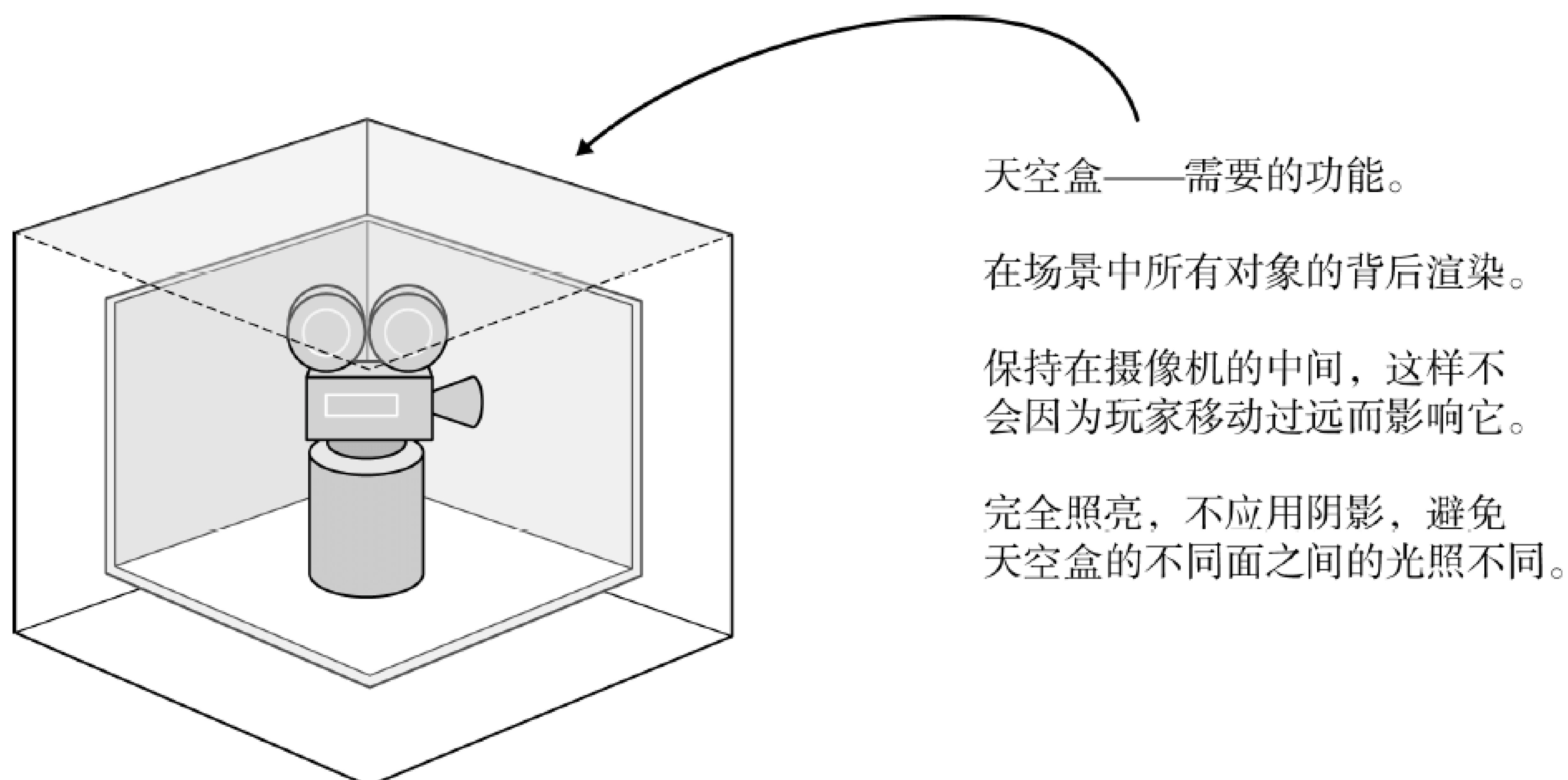


图 4-9　天空盒的图解

与之前的砖块贴图一样，可以从各种网站上获取天空盒图像。例如，搜索 skybox textures，可以从 http://93i.de/上获取大量天空盒图像，包括 TropicalSunnyDay 系列。将这个天空盒应用到场景，结果如图 4-10 所示。

与其他贴图一样，天空盒图像首先赋予材质，然后在场景中使用这个材质。接下来介绍如何创建新天空盒材质。



图 4-10　有天空背景图片的场景

4.4.2　创建一个新天空盒材质

首先，创建一个新材质(像往常一样，右击并选择 Create 或者从 Assets 菜单中选择 Create)，选择刚创建的材质，在 Inspector 中查看其设置。接下来需要改变这个材质使用的着色器(shader)。材质设置的顶部有 Shader 菜单(如图 4-11 所示)。本章的 4.3 节会

忽略这个菜单，因为默认选项适合于大多数标准贴图，但天空盒需要一个特殊的着色器。

定义　着色器是一种简短的程序，它列出了绘制表面的指令，包括是否使用贴图。计算机在渲染图像时使用这些指令来计算像素。最常见的着色器使用材质的颜色，根据灯光决定明暗，但着色器也能用于实现各种视觉效果。

每个材质都有一个控制它的着色器(可以认为每种材质都是着色器的一个实例)。新材质默认设置了标准着色器。在表面上应用基础的明暗时，该着色器会显示材质(包括贴图)的颜色。

天空盒使用另一种着色器。单击菜单，观察下拉列表(如图 4-11 所示)中所有可见的着色器。移动到 Skybox 部分，并在子菜单中选择“6 Sided”。

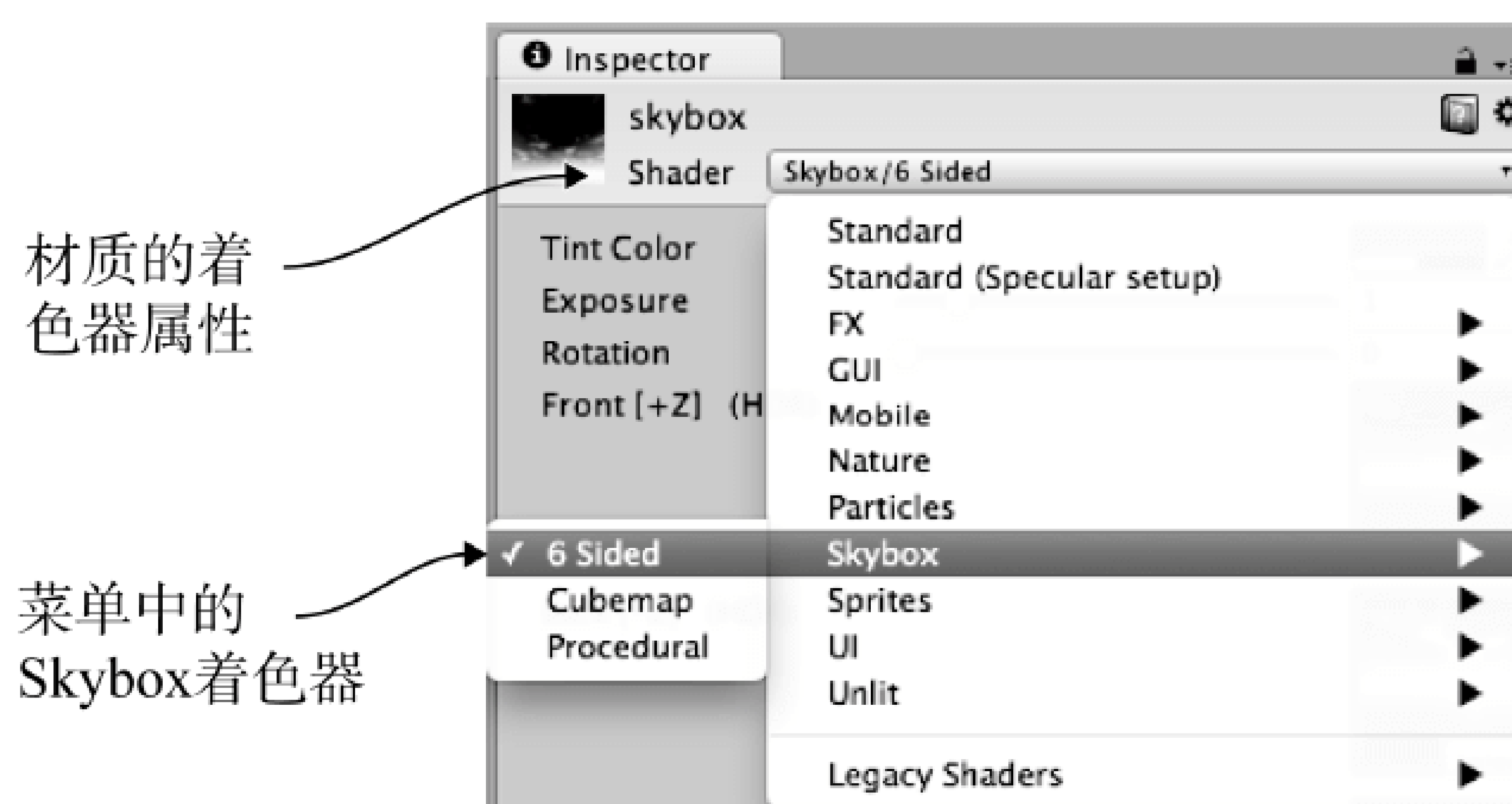


图 4-11　下拉菜单中的可见着色器

启用这个着色器，材质现在有 6 个大的贴图槽(而不是标准着色器中小的 Albedo 贴图槽)。这 6 个贴图槽对应正方体的 6 个面，因此这些图像的边缘应相匹配，以实现无缝衔接。例如，如图 4-12 所示是用于晴天天空盒的图像。

从93i.de下载的天空盒图像：上、下、前、后、左、右

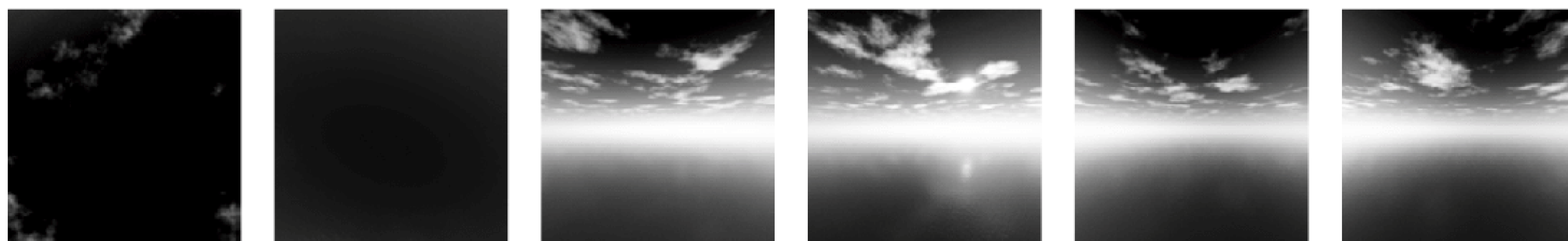


图 4-12　天空盒的六面——用于上、下、前、后、左、右的图像

将天空盒图像导入Unity的方式与导入砖块贴图一样：将文件拖动到Project视图中，或在 Project 中右击，并选择 Import New Asset。导入设置只需要一个修改，单击导入的贴图，在 Inspector 中查看它的属性，并将 Wrap Mode 设置(如图 4-13 所示)从 Repeat 修改为 Clamp(不要忘记在修改后单击 Apply)。通常贴图可以在表面重复平铺。为了实现无缝衔接，图像的对边需要衔接起来。但这种边缘混合会在天空的图像交接处产生模糊的线条，因此 Clamp 设置(类似第 2 章的 Clamp()函数)会限制贴图的边界，去除这种混合。

现在可以将这些图像拖动到天空盒材质的贴图槽。图像名称对应它们赋值到的贴图槽名称(例如 left 或 front)。六个贴图连接上贴图槽之后，就可以使用新材质作为场景中的天空盒。重新打开 lighting 窗口，把新材质设置到 Skybox 槽，可以将材质拖动到 Skybox 槽上，或单击小圆圈图标，打开文件拾取器。

提示 默认情况下，Unity 将在编辑器的 Scene 视图中显示天空盒(或者至少显示它的主色)。当编辑对象时，这种颜色可能会分散注意力，因此可以将天空盒切换为开启或关闭状态。在 Scene 视图顶部的窗格里，有控制是否显示天空盒的按钮；查找用于切换天空盒开启或关闭的 Effects 按钮。

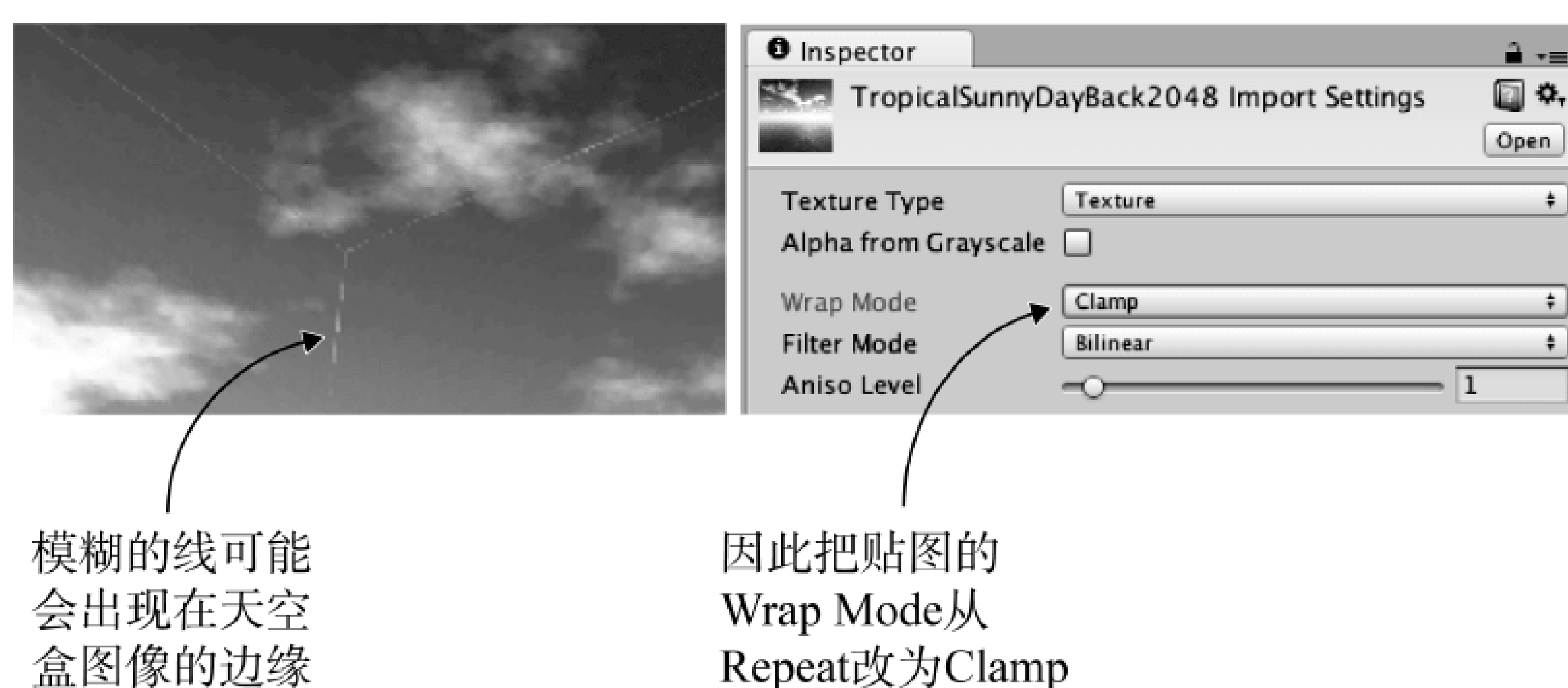


图 4-13　通过调整 Wrap mode 来改变边缘模糊的线条

这就学会了如何为场景创建天空视觉效果！天空盒是创建包围玩家的大环境的一种优雅方式。打磨关卡中的视觉效果的下一步是创建更复杂的 3D 模型。

4.5　使用自定义 3D 模型

前面章节讨论了如何将贴图应用到关卡中大而平坦的墙壁和地板上。带有更多细节的物件该怎么办？如果房间中有几件有趣的家具，该怎么办？为此，可以通过外部 3D 美术应用程序建立 3D 模型。回想本章引言的定义：3D 模型是游戏中的网格对象(即三维形状)。接下来导入一个简单长凳的 3D 模型。

主要用于 3D 对象建模的应用程序有 Autodesk 的 Maya 和 3ds Max。这些都是昂贵的商业化工具，因此本章的示例采用开源软件 Blender。示例下载中包括了可以使用的.blend 文件，图 4-14 展示了 Blender 中的长板凳模型。如果希望学习如何给自己的对象建模，可以在附录 C 中找到在 Blender 中建模这个长板凳的练习。

这包括了3D网格几何体和
应用到网格上的贴图

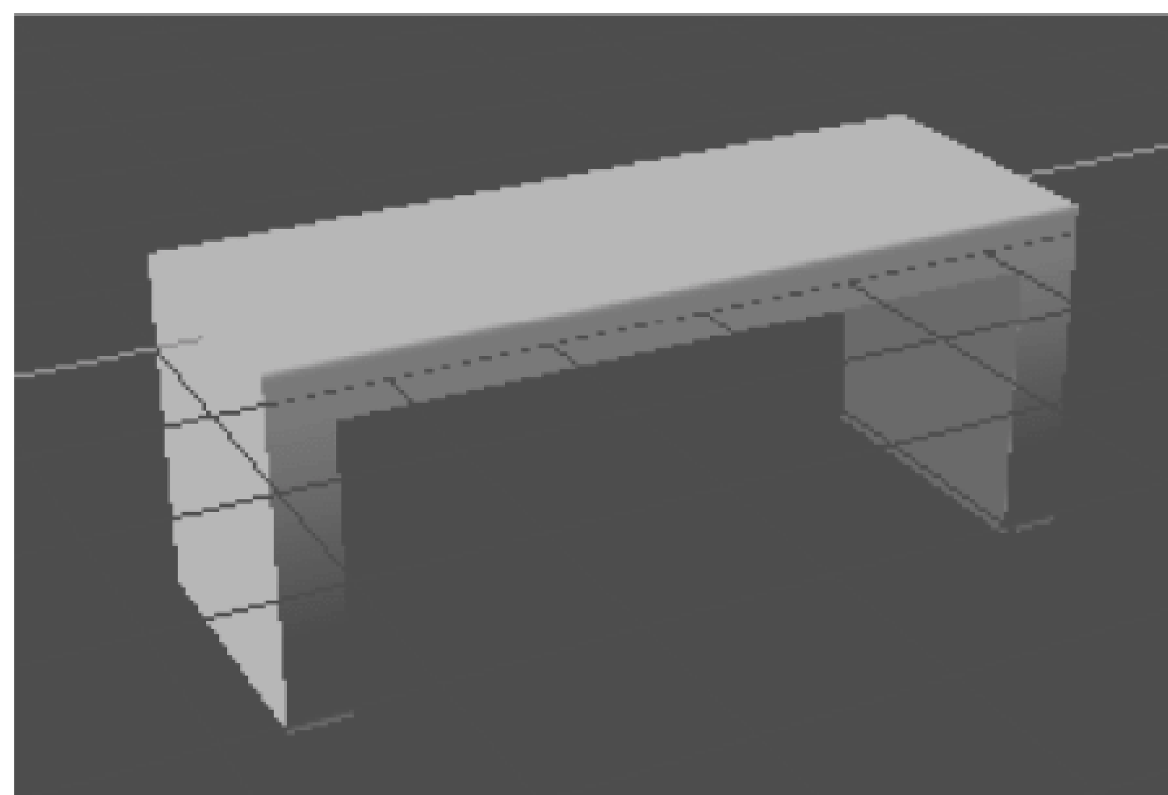


图 4-14　Blender 中的长板凳模型

除了自己或一起工作的设计师定制的模型外，还可以从游戏美术资源网站下载很多 3D 模型。一个下载 3D 模型的资源网站是 Unity 的 Asset Store，网址为 https://www.assetstore.unity3d.com。

4.5.1　选择文件格式

得到了在外部美术工具中制作的模型后，就需要从这个软件导出资源。像 2D 图像一样，当导出 3D 模型时，有很多不同的文件格式可以使用，而这些文件格式有各种优缺点。表 4-3 列出了 Unity 支持的 3D 文件格式。

表 4-3　Unity 支持的 3D 模型文件格式

文 件 类 型	优　缺　点
FBX	网格和动画；若可以使用这种文件格式，则推荐使用
Collada(DAE)	网格和动画；当 FBX 文件格式不可用时，该文件格式是一个不错的选择
OBJ	只有网格；这是文本格式，因此有时用于互联网上的传输流
3DS	只有网格；比较老的模型格式
DXF	只有网格；比较老的模型格式
Maya	通过 FBX 工作；需要安装 Maya 软件
3ds Max	通过 FBX 工作；需要安装 3ds Max 软件
Blender	通过 FBX 工作；需要安装 Blender 软件

选择哪个文件格式取决于该文件是否支持动画。由于只有 Collada 和 FBX 包含动画数据，因此只能选择这两个选项。只要 FBX 导出选项可用(不是所有的 3D 工具都支持导出 FBX 选项)，就使用该选项。但如果使用的工具不能导出 FBX，也可以使用 Collada。在本例中，Blender 支持导出 FBX，因此这里使用这种文件格式。

注意，表 4-3 的底部列出了一些 3D 美术应用程序。Unity 允许直接将这些应用程序的文件拖到项目中，最开始会觉得很方便，但这个功能有一些要注意的问题。首先 Unity 不是直接加载这些应用程序的文件，而是在后台导出模型，再加载导出文件。因为模型最终都会导出为 FBX 或 Collada，所以最好明确地执行导出这一步。此外，这种导出需要安装了对应的应用程序。如果计划在多台计算机中共享这些文件(例如，开发团队一起工作)，这个要求就很难做到。不建议直接在 Unity 中使用 3D 美术应用程序的文件。

4.5.2　导出和导入模型

下面从 Blender 导出模型，再把它导入到 Unity 中。首先在 Blender 中打开长

板凳，然后选择 File | Export | FBX。保存文件后，把它导入到 Unity 中，其方式与导入图像相同。从计算机将 FBX 文件拖动到 Unity 的 Project 视图，或者在 Project 中右击并选择 Import New Asset。3D 模型会复制到 Unity 项目中，可以放入场景中了。

注意 下载的示例中包括了.blend 文件，因此可以练习从 Blender 中导出 FBX 文件；虽然不一定自己建模，但可能需要把下载的模型转换为 Unity 能接受的格式。如果想跳过涉及 Blender 的所有步骤，可以使用所提供的 FBX 文件。

有一些用于导入模型的默认设置可能需要立刻修改。首先，Unity 默认把导入模型缩到很小(图 4-15 显示了当选择模型时 Inspector 中的信息)，把 Scale Factor 改为 100，可以部分抵消 File Scale 为 0.01 的设置。还要选中 Generate Colliders 复选框，但这是可选的，没有碰撞器，可以穿过长板凳。接着在导入设置中切换到 Animations 标签，取消选择 Import Animation(这个模型不需要动画)。

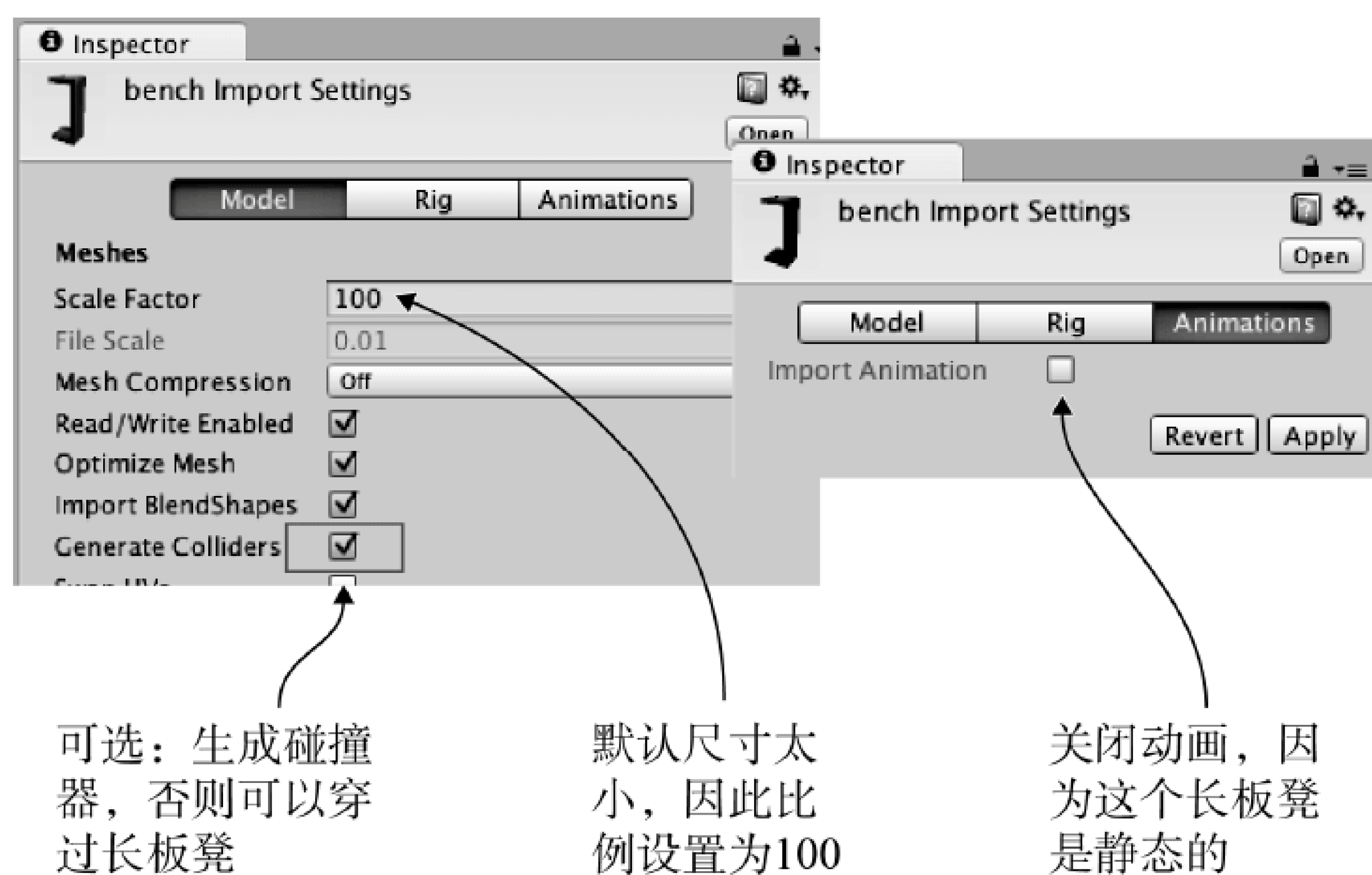


图 4-15　调整 3D 模型的导入设置

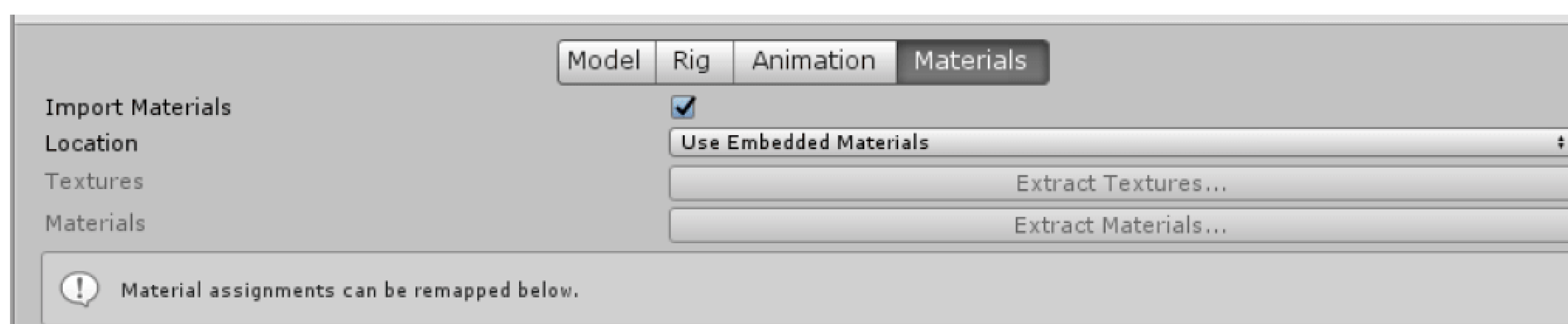
上面讨论了导入的网格，现在关注贴图。当 Unity 导入 FBX 文件时，它也为长板凳创建了一个材质。这个材质默认为空(像其他新材质一样)，因此为长板凳赋予贴图(如图 4-16 所示)，其方式与之前将砖块贴图赋给墙壁一样。将贴图图像拖动到 Project 中，将贴图导入到 Unity，接着将导入的贴图拖动到长板凳材质的贴图槽上。图像看起来有点古怪，图像的不同部分出现在长板凳的不同位置上；编辑模型的贴图坐标来定义图像到网格的映射。



图 4-16　用于长板凳贴图的 2D 图像

定义 贴图坐标是一系列额外的值，这些值用于指定贴图图像区所在的多边形的顶点。想象一下包装纸，3D 模型是将被包装的盒子，贴图是包装纸，贴图坐标表示盒子的每一边对应包装纸的哪个地方。

注意 即使不想建模长板凳，也可以在附录 C 中阅读有关贴图坐标的详细解释。贴图坐标(还有其他的相关术语，像 UV 和映射)对游戏编程十分有用。
新版 Unity 中增加了新的导入材质的方式，如果你的模型中包含了材质，也可以简单地通过单击被导入模型的 Inspector 中的 Materials 标签页，在 Location 下拉框中选择 Use Embedded Materials，并单击 Extract Materials 以导入材质。



新材质太明亮，所以需要将 Smoothness 设置减小为 0(表面越光滑，就越明亮)。最后，调整完所有需要调整的设置，就可以将长板凳放在场景中了。将模型从 Project 视图拖到关卡的一个房间中。随着鼠标的拖动，可以在场景中看到长板凳。放开鼠标时，结果如图 4-17 所示。为关卡创建了带贴图的模型！



图 4-17　导入到关卡中的长板凳

注意 通常需要使用外部工具创建的模型来替代白盒几何体，但本章不打算这样做。新几何体看起来基本一样，但可以更灵活地控制贴图。

使用 Mecanim 给角色建立动画

前面创建的模型是静态的，一直停留在放置它的位置。也可以在 Blender 中给它建立动画，在 Unity 中播放动画。创建 3D 动画是漫长的过程，但本书不介绍动画，因此不会就此进行讨论。在建模时提到，如果想要学习 3D 动画，也有很多资源。但注意，3D 动画是个巨大的话题。

Unity 中有一个称为 Mecanim 的专门系统，用于管理模型上的动画。名称 Mecanim 代表更新、更高级的动画系统，添加到 Unity 中，作为旧动画系统的替代。旧动画系统仍然存在，标识为 legacy animation。但旧动画系统将在 Unity 的后续版本中被移除，那时 Mecanim 将成为指定的动画系统。

但是本章没有使用任何动画，第 7 章将播放角色上的动画。

4.6　使用粒子系统创建效果

除了 2D 图像和 3D 模型之外，粒子系统是游戏设计师创建的另一种可视化内容。本章引言中的定义阐明了，粒子系统是一种创建和控制大量移动对象的规则机制。粒子系统可用于创建视觉效果，诸如火焰、烟雾或喷水。例如，图 4-18 是通过粒子系统创建的火焰效果。

大多数美术资源通过外部工具创建，再导入项目，而粒子系统通过 Unity 自身创建。Unity 提供了一些灵活且强大的工具来创建粒子效果。

注意　和 Mecanim 动画系统一样，过去使用旧的粒子系统，而新的系统有一个特殊的名称 Shuriken。现在，旧的粒子系统已被移除，因此这个独立的名称也不再需要。

首先，创建一个新的粒子系统并观察默认效果。从 GameObject 菜单选择 Particle System，就会看到基本的白色小球从新对象往上喷洒。更精确地说，当选中对象后，粒子就往上喷洒。在选择一个粒子系统后，在场景的角落将显示粒子回放面板，它指示过了多长时间(如图 4-19 所示)。



图 4-18　使用粒子系统创建的火焰效果



图 4-19　粒子系统的回放面板

默认效果看起来相当平滑，下面浏览一下可以用于定制效果的大量参数。

4.6.1　调整默认效果的参数

图 4-20 显示了粒子系统的完整设置列表。在此不打算解释列表中的每个设置，而是讨论火焰效果的对应设置。明白一些设置如何工作之后，其他设置就自然容易理解了。每个设置标签实际是一个完整的信息面板。最初只展开了第一个信息面板，其他面板都折叠起来了。单击设置的标签可以展开信息面板。

提示　很多设置通过显示在 Inspector 底部的曲线来控制。该曲线描述了值随时间的变化情况：图的左边表示粒子首次出现的时间，右边表示粒子消逝的时间，底部的值为 0，顶部为最大值。在图中拖动点，在曲线上双击或右击，可以插入新的点。

如图 4-20 所示调整粒子系统的参数，这个粒子系统看起来像一个发射的火焰。

Looping：粒子系统一直播放；选择默认值即可

Lifetime：粒子存在的时间；减少为3

Speed：粒子移动得多快；减少为1

Size：粒子多大；选择默认值即可

Rotation：粒子的方向；单击箭头菜单修改为Between Constants，并设置为0和180

Color：给粒子染色。我们需要暗橙色，例如RGB值为(182, 101, 58)

Local的仿真空间适合于静止的粒子系统，但当粒子系统移动时，使用World可能会更好

Emission：粒子发射的速度；选择默认值

Shape：发射区域的形状。默认是一个开口的圆锥，但我们改成一个小盒子，以造成一种紧凑的火焰喷射(设置为Box，所有数字为0.2)

Size over Lifetime：粒子在运动时变大和缩小。默认关闭；打开它并单击箭头，设置一条从0快速变大并缓慢变小为0的曲线(见本图所示)

Rotation over Lifetime：当粒子运动时旋转。默认关闭，打开它并将Random设置为-80～80之间，使不同粒子在不同方向旋转

Renderer：设置每个粒子的外观。甚至可以把它设置为网格，但这里将它设置为Billboard并拖入一个新材质(后面将解释)

双击或右击曲线并选择Add Key，在曲线上添加点

图 4-20　Inspector 显示粒子系统的设置(火焰效果的设置)

4.6.2　为火焰应用新贴图

现在粒子系统看起来更像是发射的火焰，但效果依然需要看起来像火焰，而不是白点。这需要将一个新图像导入到 Unity 中。图 4-21 是一副绘制好的图像，其中有一个橙色点并使用 Smudge 工具绘制卷曲的火焰(接着用黄色绘制了相同的形状)。无论是使用示例项目中的这个图像、拖入自己的图像还是下载类似的图像，都需要将图像文件导入到 Unity 中。如前所述，将图像文件拖动到 Project 视图或者使用 Assets | Import New Asset。



图 4-21　用于火焰粒子的图像

类似于 3D 模型，贴图没有直接应用于粒子系统。将贴图添加到一个材质上，再将该材质应用到粒子系统。创建新材质并选择它，以便在 Inspector 中显示它的属性。把 Project 中的火焰图像拖到贴图槽上。这样就将火焰贴图关联到火焰材质上了，现在需要将材质应用到粒子系统上。如图 4-22 所示进行操作，选择粒子系统，展开设置底部的 Renderer，并将材质拖动到 Material 槽上。

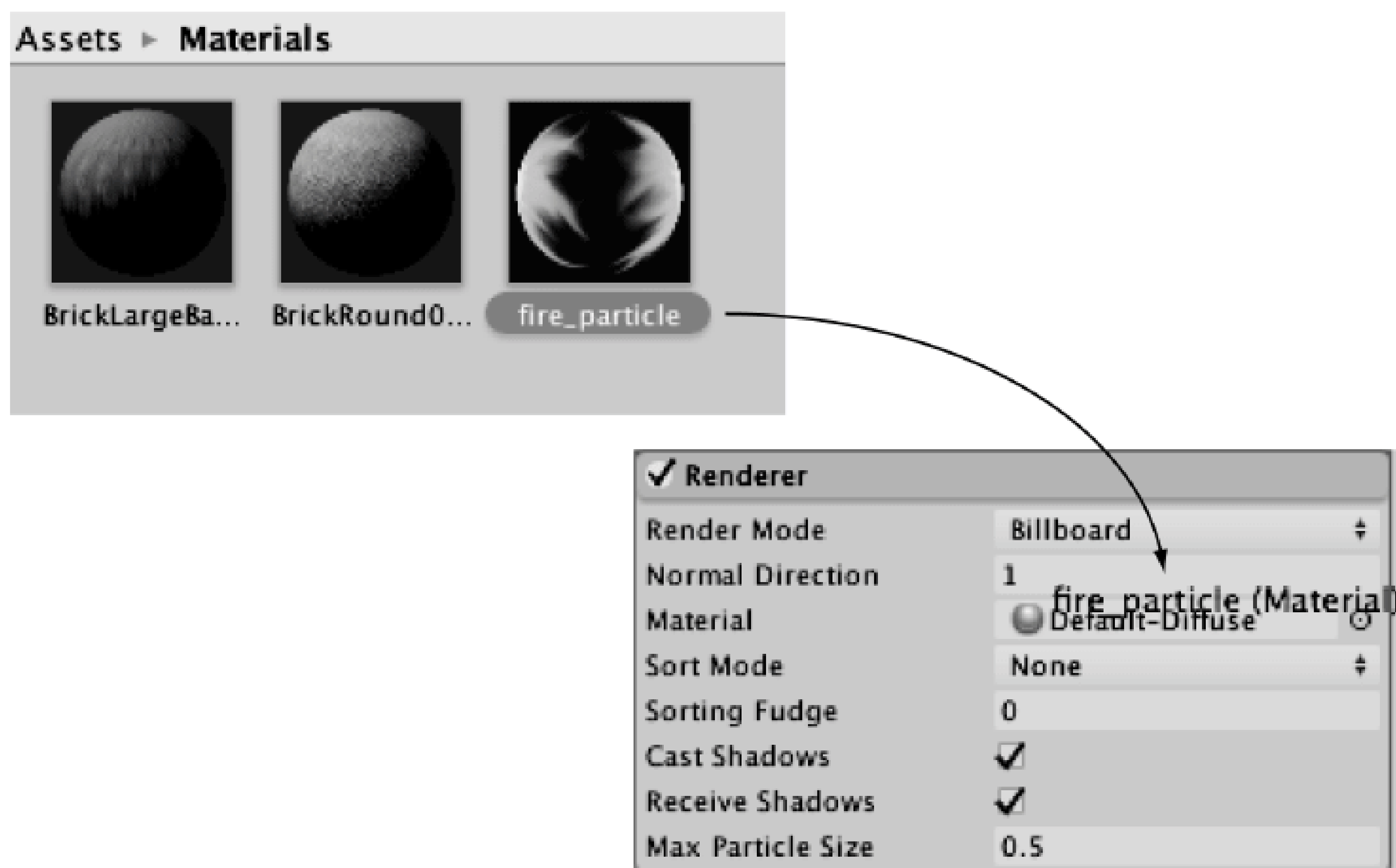


图 4-22　将材质应用到粒子系统上

与天空盒材质一样，需要修改粒子材质的着色器。单击材质设置顶部的 Shader 菜单，观察可用着色器的列表。与默认材质不同，粒子的材质需要 Particles 子菜单下的一个着色器。如图 4-23 所示，本例需要 Additive(Soft)着色器。这会让粒子在场景中变得朦胧和明亮，犹如火焰一般。

定义　Additive 是一种将粒子颜色叠加到它背后的颜色上，而不是替换像素颜色的着色器。这让像素更明亮，而粒子的黑色部分不可见。与之相对的着色器是 Multiply，它让对象变得更暗。这些着色器的视觉效果和 Photoshop 中 Additive 和 Multiply 的图层效果一样。

把火焰材质赋予火焰粒子效果，结果如图 4-18 所示，看起来更像发射的火焰，但

该效果不仅用于静止状态，接下来将它附加到一个运动的对象上。

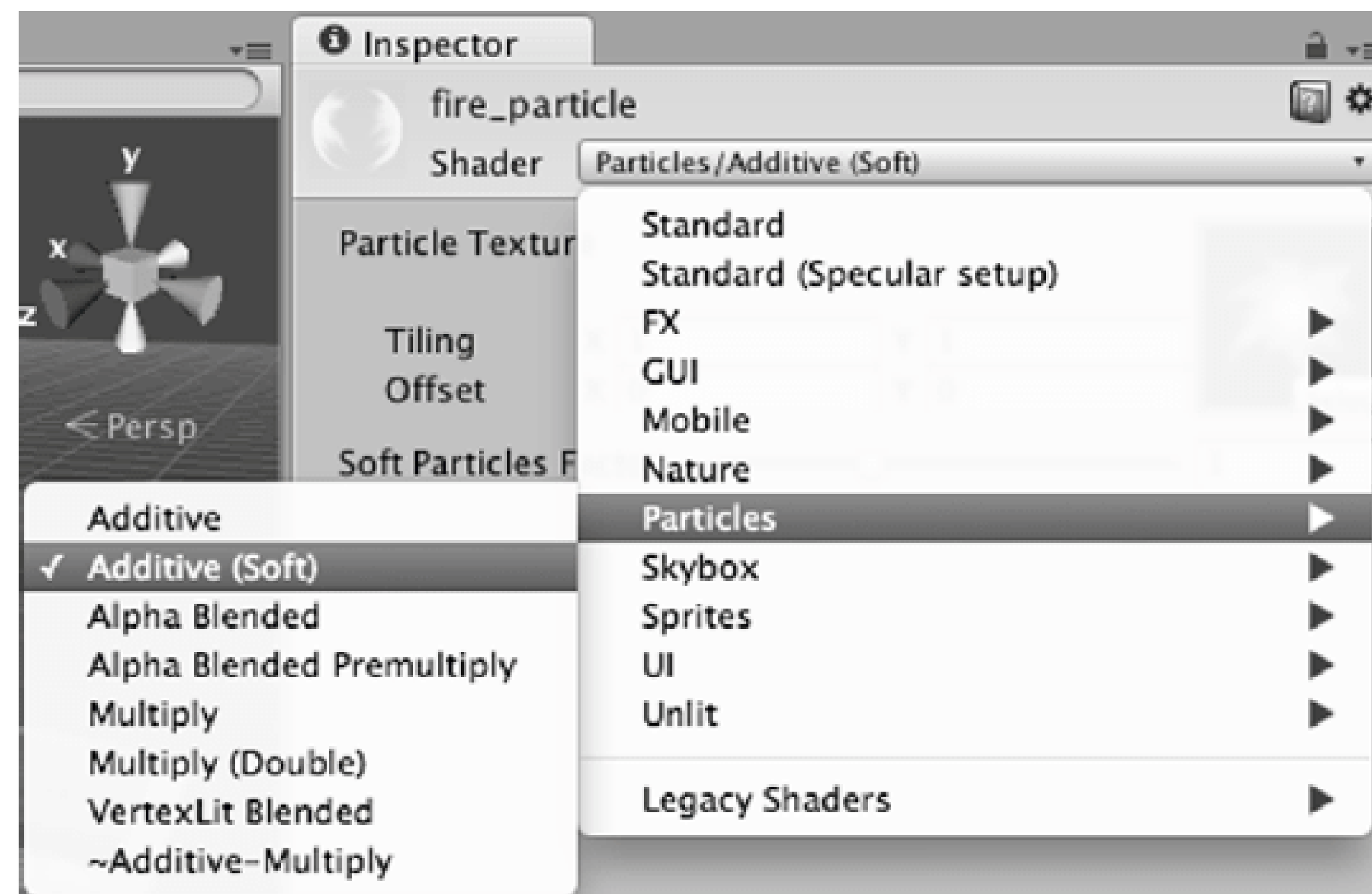


图 4-23　为火焰粒子材质设置着色器

4.6.3　将粒子效果附加到 3D 对象上

创建一个球体(选择 GameObject | 3D Object | Sphere)。新建一个名为 BackAndForth 的脚本，如代码清单 4.1 所示，并将脚本附加到新球体上。

代码清单 4.1　沿着直线路径前后移动对象

```
using UnityEngine;
using System.Collections;

public class BackAndForth : MonoBehaviour {
    public float speed = 3.0f;
    public float maxZ = 16.0f;
    public float minZ = -16.0f;

    private int _direction = 1;

    void Update(){
        transform.Translate(0, 0, _direction * speed * Time.deltaTime);

        bool bounced = false;
        if (transform.position.z > maxZ || transform.position.z < minZ) {
            _direction = -_direction;
            bounced = true;
        }
        if (bounced) {
            transform.Translate(0, 0, _direction * speed * Time.deltaTime);
        }
    }
}
```

这些是对象移动的位置范围

当前对象往哪个方向移动

切换来回的方向

如果切换方向，本帧对象额外移动一次

运行上述脚本，球体在关卡中间的走廊前后移动。现在可以让粒子系统成为球体的子结点，粒子将随着球体移动。如同处理关卡墙壁一样，在 Hierarchy 视图中将粒子对象拖动到球体对象上。

警告 通常，需要在对象成为另一个对象的子对象之后重置子对象的位置。例如，粒子系统应定位在(0, 0, 0)(这是相对它的父对象而言)。Unity 会在对象成为其子对象前保存它的位置。

现在粒子系统随着球体运动。火焰没有因为球体运动而偏转，这看起来不自然。这是因为粒子默认在粒子系统的本地坐标中移动。为了完成火球效果，找到粒子系统设置中的 Simulation Space，将它从 Local 切换为 World。

注意 在本脚本中，对象在直线上前后移动，但视频游戏中的对象通常在更复杂的路径上移动。Unity 支持复杂的导航和路径。请查阅 https://docs.unity3d.com/Manual/Navigation.html 了解它。

此时读者应渴望将自己的想法应用到游戏中，给这个示例游戏添加更多内容。可以创建更多的美术资源，甚至引入第 3 章开发的射击机制，来测试所掌握的技能。下一章会开始一个新游戏，切换到另一种游戏种类，但本书前 4 章的内容依然能应用其中并发挥作用。

4.7　小结

- 美术资源是表示所有图形的术语。
- 白盒是关卡设计师用于分隔出空间的第一步。
- 贴图是显示在 3D 模型表面上的 2D 图像。
- 3D 模型在 Unity 外部创建并作为 FBX 文件导入。
- 粒子系统用于创建很多可视化效果(火焰、烟雾、水等)。

第 Ⅱ 部分

轻 松 工 作

前面在 Unity 中构建了第一个游戏原型，现在准备处理其他游戏类型以扩展基础知识。目前，使用 Unity 工作的步骤很相似：创建包含各种功能的脚本，将对象拖到 Inspector 的槽上等。你不再被操作界面的细节所困扰，这意味着剩余章节不再需要提及基础知识。

接下来完成其他一系列项目，逐步提升在 Unity 中开发游戏的能力。

第 5 章

使用 Unity 的 2D 功能构建一款记忆力游戏

本章涵盖：

- 在 Unity 中显示 2D 图形
- 使对象可以单击
- 通过编程加载新图像
- 使用 UI 文本管理和显示状态
- 加载关卡和重新开始游戏

前面一直在处理 3D 图形。其实也可以在 Unity 上使用 2D 图形，因此本章将学习如何构建 2D 游戏。我们将开发一款经典的儿童记忆力游戏：游戏将显示一些卡背，当单击时显示正面，卡片匹配则记录分数。这些技术涵盖了在 Unity 中开发 2D 游戏必知的一些基础知识。

尽管 Unity 作为 3D 游戏工具而生，但它也可以用于 2D 游戏。Unity 的版本(从 4.3 版本开始)增加了显示 2D 图形的功能，但之前已经有使用 Unity 开发的 2D 游戏(特别是移动游戏，它受益于 Unity 跨平台的特性)。在之前的 Unity 版本中，游戏开发者需要第三方框架(例如 Unikron Software 的 2D Toolkit)，在 Unity 的 3D 场景中模拟 2D 图形。最终，核心编辑器和游戏引擎已修改为包含 2D 图形，而本章将介绍这些新功能。

Unity 中的 2D 工作流或多或少和开发 3D 游戏的工作流一样：导入美术资源，将它们拖动到场景，编写脚本附加到对象上。2D 图形中主要的美术资源类型称为精灵(sprite)。

定义 精灵是直接显示在屏幕上的 2D 图像，和显示在 3D 模型表面的图像不同(那称为贴图)。

可以采用与导入图像作为贴图(参阅第 4 章)一样的方式将 2D 图像导入到 Unity 中作为精灵。从技术上讲，这些精灵是 3D 空间的对象，但它们是扁平的，且垂直于 Z 轴。由于它们面对相同方向，因此可以让摄像机直接面对精灵，而玩家只能沿着 X 和 Y 轴移动(这就是二维)。

在第 2 章中讨论了坐标轴：三维是加入了同时垂直于 X 轴和 Y 轴的 Z 轴。二维则只有 X 轴和 Y 轴。

5.1　设置 2D 图形

下面将创建经典的记忆力游戏。对于不熟悉这款游戏的人而言，在该游戏中，一些卡片先设置为面向下。每张卡片都会在某个地方有一张匹配的卡片，但是玩家只能看到该卡片的背面。玩家能一次翻开两张卡片，尝试找到相匹配的卡片；如果选中的两张卡片不匹配，这两张卡片会再次翻转回去，让玩家继续猜。

图 5-1 展示了要构建的游戏原型；把图 5-1 与第 2 章的路线图进行对比。

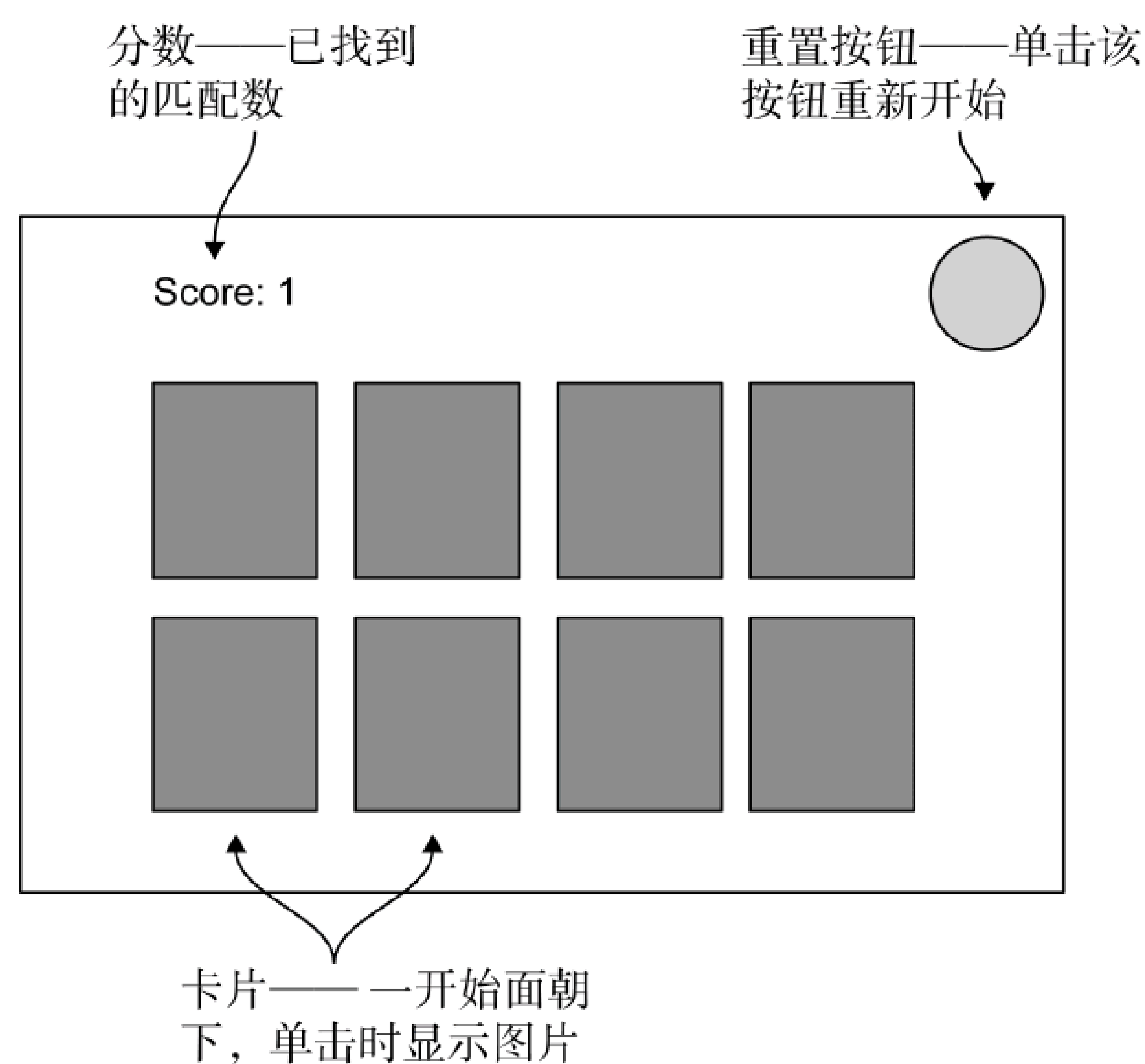


图 5-1　记忆力游戏的原型

注意，该原型此时描绘了玩家看到的画面(而 3D 场景原型描述了玩家周围的空间和用于玩家观察场景的摄像机位置)。知道了要构建什么场景，就该开始工作了！

5.1.1　为项目做准备

第一步是为游戏收集和显示图形。与之前构建 3D 演示游戏的方式大同小异，需

要在开发新游戏之前，准备好游戏操作所需的最小图形集合。在这些工作完成后，就可以开始编写游戏功能。

这意味着需要创建如图 5-1 所示的对象：用于隐藏卡片的卡背、当卡片翻过来时的若干卡片正面、显示在角落的分数、显示在另一个角落的重置按钮。场景还需要一个背景，因此下面将所有美术需求整理在图 5-2 中。

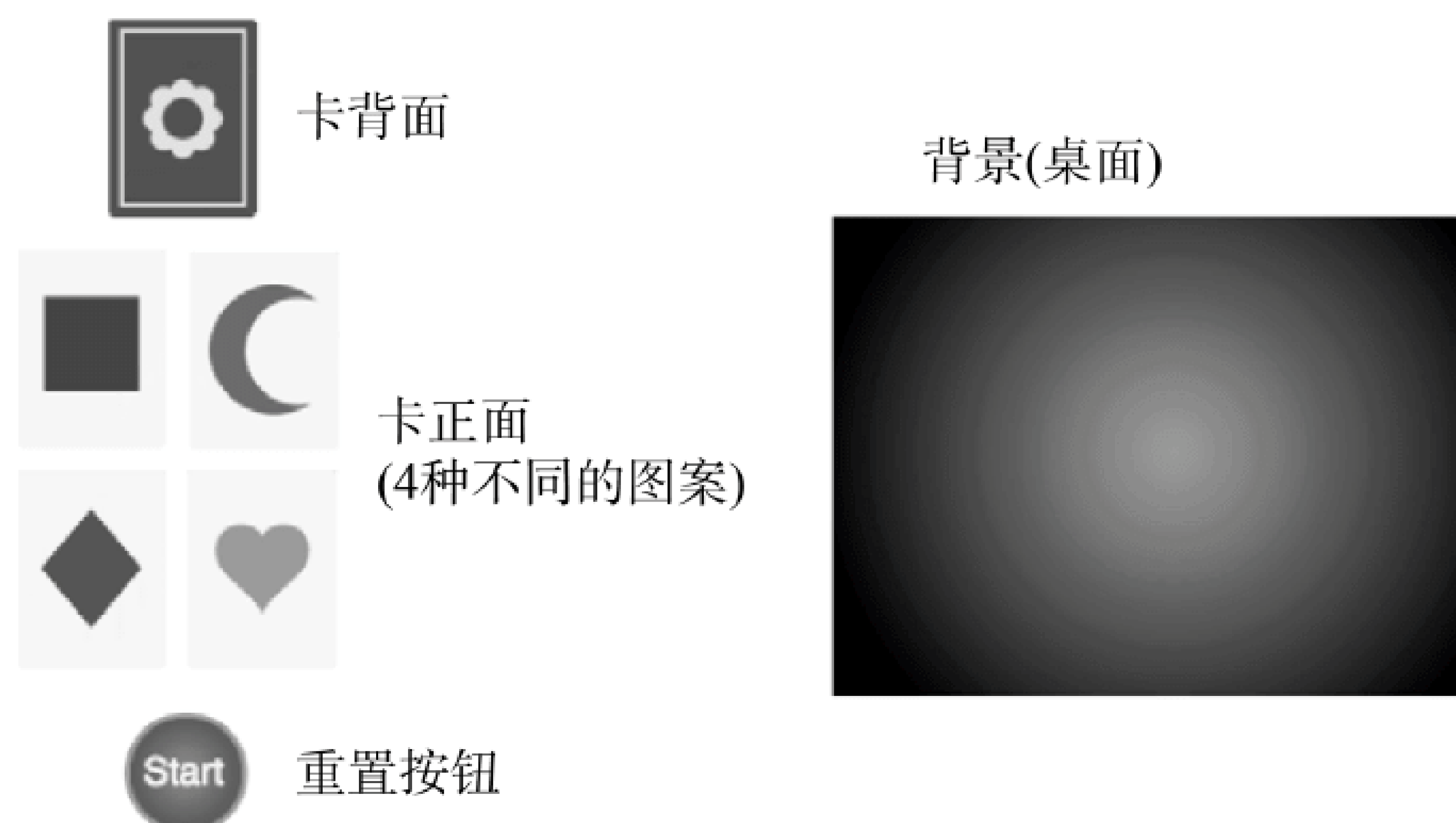


图 5-2　记忆力游戏所需的美术资源

提示　和前面一样，本项目的完成版本，包括所有需要的美术资源，都可以从本书的网站 www.manning.com/books/unity-in-action-second-edition 上下载。可以将这些图像复制到自己的项目中。

收集所需的图像，接着在 Unity 中创建新项目。在出现的 New Project 窗口中，注意底部的一些按钮(如图 5-3 所示)可以在 2D 和 3D 模式间进行切换。前几章处理的是 3D 图形，而 3D 是默认模式，因此我们没有关注这个设置。然而本章将在创建新项目时切换到 2D 模式。



图 5-3　使用这些按钮确定是以 2D 还是 3D 模式创建新项目

2D Editor 模式和 2D Scene 视图

新项目的 2D/3D 设置调整了 Unity 编辑器中两个不同的设置，这两个设置都可以在以后手动调整。这两个设置是 2D Editor 模式和 2D Scene 视图。2D Scene 视图控制在 Unity 中如何显示场景；可以切换 Scene 视图顶部的 2D 按钮。



设置 2D Editor 模式时，可以打开 Edit 菜单，并选择 Project Settings 下拉菜单中的 Editor。在这些设置中，Default Behavior Mode 设置有 3D 或 2D 选项。



2D/3D Behavior Mode菜单

Edit | Project Settings | Editor中的 Default Behavior Mode设置

将编辑器设置为 2D 模式，会将导入的图像设置为 Sprite，如第 4 章所示，通常图像导入为贴图。2D 编辑器模式也使新场景缺乏默认的 3D 光源设置，这不会影响 2D 场景，但它并不是必需的。如果需要手动移除它，删除新场景中的平行光源，并在 lighting 窗口中关闭天空盒(单击小圆圈图标，打开文件选择器，并从列表中选择 None)。

在为本章创建新项目并设置为 2D 之后，可以开始将图像放到场景中。

5.1.2　显示 2D 图像(亦称精灵)

将所有图像文件拖动到 Project 视图，以导入它们，确认图像被导入为精灵而不是贴图(如果将编辑器设置为 2D 模式，就会自动导入为精灵。选择一个资源，在 Inspector 中查看它的导入设置)。现在从 Project 视图将 table_top(背景图像)精灵拖动到空场景中，并保存场景。同网格对象一样，Inspector 中包含精灵的 Transform 组件。输入(0, 0, 5)来定位背景图像。

提示　另一个需要关注的导入设置是 Pixels-To-Units。由于 Unity 最开始为 3D 引擎，而 2D 图形后来才加入，因此 Unity 中的一个单位不一定是图像中的一个像素。可以设置 Pixels-To-Units 为 1∶1，但建议使用默认的 100∶1(因为物理引擎在 1∶1 的情况下不能正常工作，默认设置也能更好地兼容别人的代码)。

使用 Sprite Packer 创建图集

虽然这个项目使用单独的图像，但是可以在单个图像中放置多个精灵。当动画的许多帧组合成一张图片时，这张图片通常称为精灵表，但是将多个图片组合成一张图片的专业术语是图集。

动画精灵在 2D 游戏中很常见，这些将在下一章中实现。可以将多个帧导入为多个图像，但游戏通常会将所有动画帧放在一个精灵表中。基本上，所有独立的帧都以网格的形式显示在一个大图像上。

除了把动画帧组合在一起之外，精灵图集也常用于静态图像。因为图集可以在两个方面优化精灵的性能：①把它们紧凑地打包在一起，减少空间的浪费，②减少视频卡的调用(每加载一个新图像将导致视频卡多做一些工作)。

可以使用 TexturePacker(见附录 B)等外部工具创建精灵地图集，这种方法肯定有效。Unity 包括一个 Sprite Packer，它可以自动将多个精灵打包在一起。要使用此功能，请在 Editor 设置中启用 Sprite Packer(在 Edit > Project Settings 下)。现在，在查看精灵图像的导入设置时，在 Packing Tag 选项中输入一个名称；Unity 将自动把拥有相同 Packing Tag 的精灵打包到同一个图集上。更多相关信息可以查阅 Unity 的文档：http://docs.unity3d.com/Manual/SpritePacker.html。

显然，位置 X 和 Y 为 0(精灵将填充整个场景，因此需要让它位于中心)，但 Z 轴位置为 5 看起来有点奇怪。对于 2D 图形，不应该只关心 X 和 Y 吗？X 和 Y 是在 2D 屏幕上影响对象定位的唯一坐标；然而 Z 坐标依然影响对象的堆叠。Z 值越低，离摄像机越近，因此 Z 值越低的精灵显示在其他精灵上(见图 5-4)，背景精灵的 Z 值应该最高。将背景设置为正的 Z 轴位置，并让其他精灵的 Z 轴为 0 或负数。

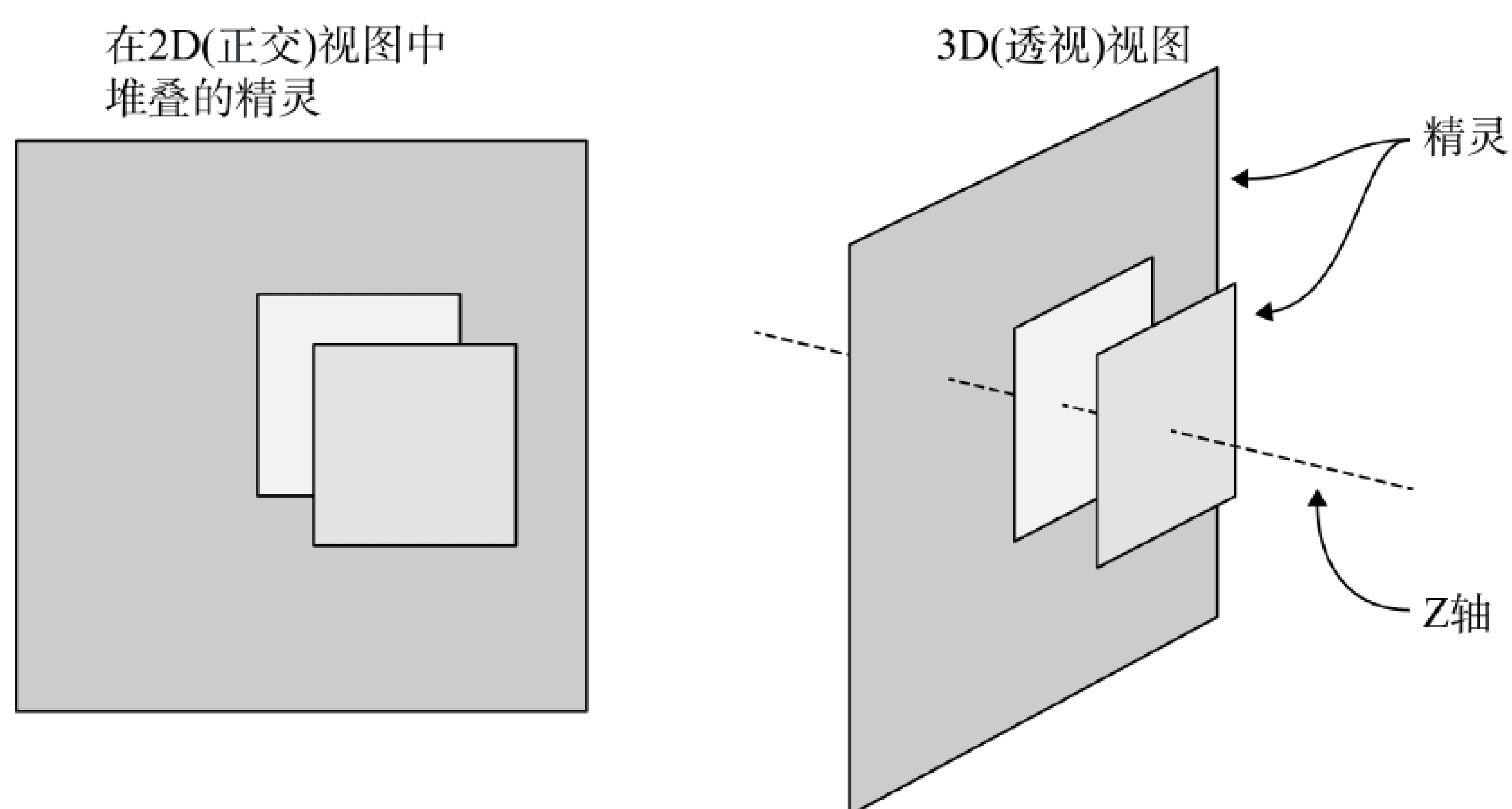


图 5-4　精灵如何沿着 Z 轴堆叠

由于之前提到的 Pixels-To-Units 设置，其他精灵的位置由 X 和 Y 值的两位小数决定。比例 100∶1 意味着图像上的 100 像素就是 Unity 中的 1 个单位。换句话说，1 个像素是 0.01 单位。接下来在将更多的精灵放到场景中之前，先设置游戏的摄像机。

5.1.3　将摄像机切换为 2D 模式

现在调整场景中的主摄像机。你可能会认为，因为 Scene 视图设置为 2D，所以在 Unity 中看到的效果将和游戏中看到的一样，这不大直观，然而事实并非如此。

警告 不管 Scene 视图是否设置为 2D，对正在运行的游戏中的摄像机视图都没有影响。

事实是不管 Scene 视图是否设置为 2D 模式，游戏中摄像机的设置都是独立的。这在很多情况下都很方便，可以将 Scene 视图切换为 3D 来处理场景中的一些效果。这种场景视图和游戏摄像机视图的拆分意味着，在 Unity 中看到的效果不一定与在游戏中看到的一样，而初学者很容易忘记这一点。

要调整的摄像机设置中最重要的是 Projection(投影)。摄像机的投影可能是正确的，因为新项目是以 2D 模式创建的，但了解并再次检查该设置依然很重要。在 Hierarchy 中选择摄像机，在 Inspector 中观察它的设置，接着查找 Projection 设置(如图 5-5 所示)。对于 3D 图形，这个设置应该是 Perspective；但对于 2D 图形，摄像机的投影应该是 Orthographic。

定义 Orthographic 是一个用于表示没有透视的平面摄像机视图的术语。它与 Perspective 摄像机相反，离 Perspective 摄像机越近的物体越大，距离摄像机越远，对象就减小。

尽管 Projection 模式是 2D 图形中最重要的摄像机设置，但还有其他一些设置也需要调整。接下来看看 Size 设置，该设置在 Projection 的下方。摄像机的 Orthographic 大小决定了摄像机视图从屏幕中心到屏幕顶部的大小。换句话说，将 Size 设置为所需屏幕像素的一半。如果将发布游戏的分辨率和像素的大小设置为相同，将得到像素完美的图形。

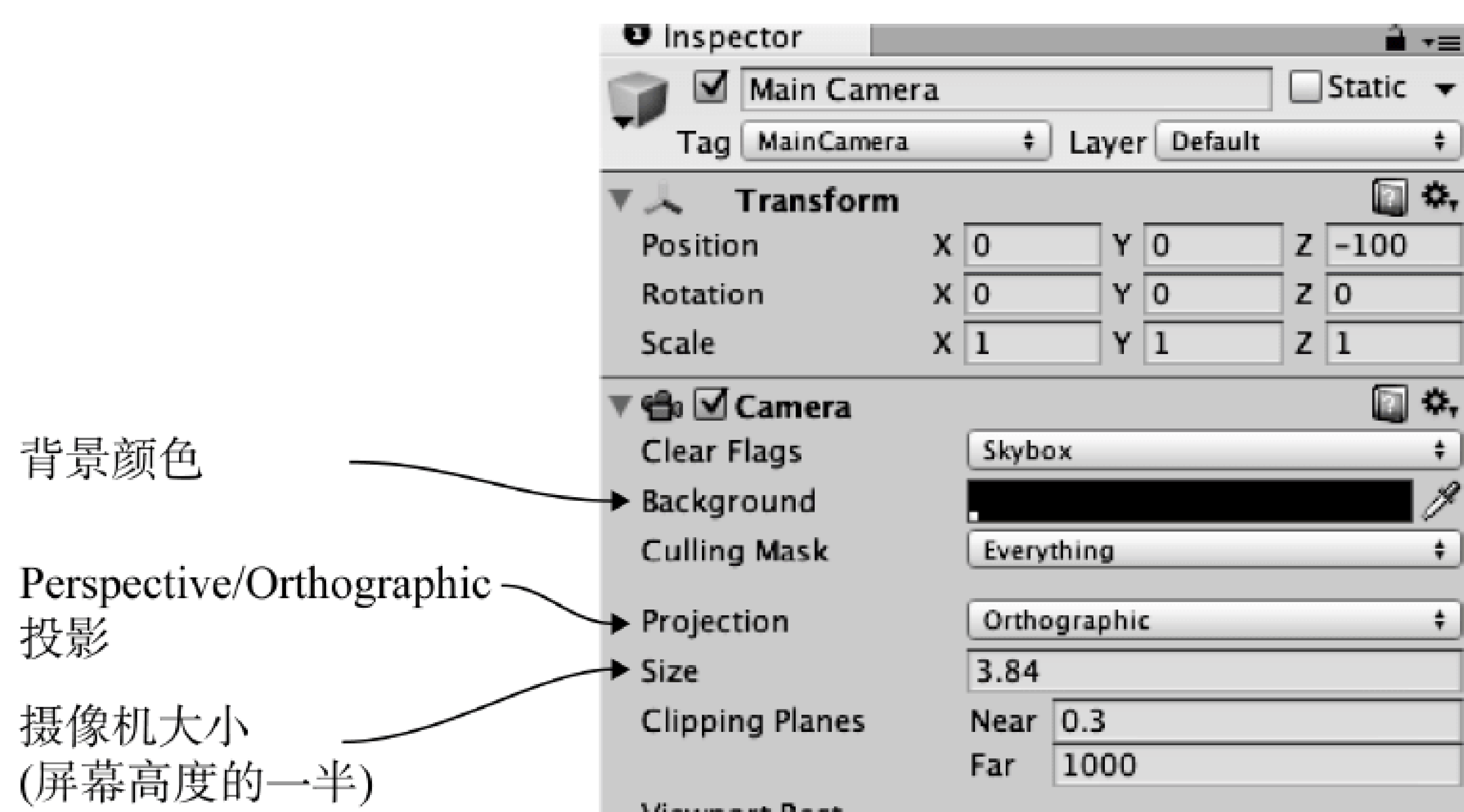


图 5-5　为 2D 图形调整摄像机设置

定义 像素完美(Pixel-perfect)意味着屏幕上的一个像素对应图像中的一个像素(否则，视频卡会让图像在缩放到填满屏幕时变得模糊)。

例如，假设要在 1024×768 屏幕上实现像素完美。这意味着摄像机的高度应该是 384 像素。除以 100(因为 Pixels-To-Units 设置)，就得到摄像机大小为 3.84。再一次声明，数学公式是 SCREEN_SIZE / 2 / 100f(f 表明是浮点数，而不是整型值)。如果背景图像是 1024×768(选择资源时选中其尺寸复选框)，则显然需要的摄像机大小是 3.84。

在 Inspector 中，剩余两处需要调整的是摄像机的背景颜色和 Z 轴位置。如之前对精灵的描述所示，更高的 Z 轴位置意味着距离场景越远，摄像机应该有更低的 Z 坐标。设置摄像机的位置为(0, 0, −100)。摄像机的背景颜色应该为黑色。默认颜色为蓝色。当屏幕比背景图像宽时，这看起来将很奇怪。单击 Background 旁边的颜色板，并通过颜色拾取器设置为黑色。

现在保存场景为 Scene，并单击 Play 按钮，桌面精灵就会填充 Game 视图。但桌面还完全是空的，因此接下来将一张卡片放在桌面上。

5.2　构建卡片对象并使它响应单击

现在导入了所有的图像，可以使用了，接下来构建这个游戏中核心的卡片对象。在记忆力游戏中，所有的卡片最初都是正面朝下，仅当玩家选择翻开一对卡片时，它们才临时正面朝上。为此，要创建由多个精灵堆叠中一起的对象。接着编写代码，使这些卡片在用鼠标单击时翻开来。

5.2.1　从精灵中构建对象

将一个卡片对象拖动到场景中。使用一个卡片正面，因为要在上面增加一个卡的背面来隐藏图像。从技术上讲，其位置目前并不重要，但最终位置还是会有影响，因此将卡片定位在(−3, 1, 0)处。现在将 card_back 精灵拖动到场景中，使这个精灵成为之前卡片精灵的子节点(记住，在 Hierarchy 中将子对象拖动到父对象上)，然后设置它的位置为(0, 0, −1)(记住这个位置是相对父节点的，所以这意味着“让它的 X 轴和 Y 轴一样，但是 Z 轴更靠近摄像机”)。

提示　在 3D 中使用 Move、Rotate 和 Scale 工具操作对象，而在 2D 模式中只使用一个称为 Rect 的工具。在 2D 模式中，这个工具会自动选中，或者可以单击 Unity 左上角最右边的导航按钮。当激活这个工具时，在二维中单击和拖动对象可以完成这三个操作(移动/旋转/缩放)。

放置卡片的背面后，如图 5-6 所示，图形就会显示能响应单击的卡片了。



图 5-6　在 Hierarchy 中链接和定位背面卡片精灵

5.2.2　鼠标输入代码

为了在玩家单击卡片时进行响应，卡片精灵需要有碰撞器组件。新的精灵默认没有碰撞器，因此它们不能被单击。接下来将一个碰撞器附加到卡片对象的根节点，而不是附加到卡片背面，因此只有卡片正面而不是卡片背面才能接收鼠标单击。为此，在 Hierarchy 中选择卡片对象根节点(不要在场景中单击来选择对象，因为卡片背面在卡片正面之上，如果在场景中单击对象，选择的就是卡片背面，而不是卡片正面)，然后单击 Inspector 中的 Add Component 按钮。选择 Physics 2D(不是 Physics，因为 Physics 是 3D 物理系统，而这个示例是 2D 游戏)，然后选择一个盒子碰撞器(box collider)。

除了碰撞器，卡片还需要一个脚本，才能对玩家的单击做出响应，因此需要编写一些代码。创建一个新的脚本 MemoryCard.cs，并将这个脚本附加到卡片对象根节点上(同样不是卡片背面)。代码清单 5.1 展示了当单击卡片时发出调试消息的代码。

代码清单 5.1　当单击时发出调试消息

```
using UnityEngine;
using System.Collections;

public class MemoryCard : MonoBehaviour {
   public void OnMouseDown() {
      Debug.Log("testing 1 2 3");
   }
}
```

单击对象时调用这个函数

现在只是发出一个测试消息到控制台

提示　如果还没有这个习惯，最好养成将资源组织到独立的文件夹中的好习惯。为脚本创建文件夹，并在 Project 视图中拖动文件。要小心避免使用 Unity 提供的特殊文件夹名称：Resources、Plugins、Editor 和 Gizmos。在本书后续章节将介绍这些特殊文件夹的作用，但现在只要避免使用这些关键字来命名文件夹即可。

现在可以单击卡片了。就像 Update()函数，OnMouseDown()是 MonoBehaviour 提供的另一个函数，它在对象被单击时响应。运行游戏，并观察显示在控制台的消息。但将消息打印到控制台仅是为了测试，接下来处理卡片正面的显示。

5.2.3　当单击时显示卡片正面

输入代码清单 5.2 中所示的代码(这些代码还不能运行，但先不必担心)。

代码清单 5.2　当卡片被单击时隐藏卡片背面的脚本

```
using UnityEngine;
using System.Collections;

public class MemoryCard : MonoBehaviour {
   [SerializeField] private GameObject cardBack;

   public void OnMouseDown() {
      if (cardBack.activeSelf) {
          cardBack.SetActive(false);
      }
   }
}
```

Inspector 中出现的变量

只是在对象当前激活/可见的情况下才运行取消激活(deactivate)代码

设置对象为不激活/不可见

这段代码有两个关键补充：引用场景中的对象和取消激活该对象的 SetActive()方法。第一点，引用场景中的对象，类似于之前章节中的处理。标记变量为可序列化，把对象从 Hierarchy 拖到 Inspector 的变量上。在设置对象引用之后，这段代码就可以影响场景中的对象。

代码中的第二个补充点是 SetActive 命令。这个命令将取消激活任何 GameObject，使对象不可见。如果现在将场景中的 card_back 拖动到 Inspector 中的脚本变量上，当运行游戏时，card_back 将在单击卡片时消失，隐藏卡片背面就会显示卡片正面。这就已经完成了记忆力游戏中的另一个重要的任务！但这仅仅是一张卡片，接下来创建一叠卡片。

5.3　显示不同的卡片图像

前面编写了一个卡片对象，它最初显示卡片背面，但当单击时则显示卡片正面。那只是一个卡片，但游戏需要一叠卡片，卡片上的图像大多不同。接下来将使用前面章节介绍的一些概念和还未讨论的几个概念来实现一叠卡片。第 3 章包括了两个概念：①使用不可见的 SceneController 组件，②实例化对象的克隆体。这次 SceneController 将不同图像应用到不同的卡片上。

5.3.1　通过编程加载图像

当前创建的游戏中包含四种卡片图案。桌面上的所有八张卡片(每种图案对应两张卡片)会通过克隆相同的源对象创建，因此所有的卡片最初都有相同的图案。我们将在脚本中改变卡片的图案，通过编程加载不同的图案。

为了说明图案如何通过编程指定，下面编写一些简单的测试代码(测试后会被替换)来演示这项技术。首先将代码清单 5.3 的代码添加到 MemoryCard 脚本中。

代码清单 5.3　演示修改精灵图像的测试代码

```
...
[SerializedField] private Sprite image;
void Start() {
    GetComponent<SpriteRenderer>().sprite = image;
}
...
```

引用要加载的精灵资源

设置这个SpriteRenderer组件的 sprite 属性

当保存这段脚本后，新的 image 变量将显示在 Inspector 中，因为 image 变量已设置为 serialized。从 Project 视图(选择一个卡片图像，但不要选择场景中已有的图像)中拖动精灵并在 Image 槽上释放。现在运行此场景，新图像就出现在卡片上。

理解这段代码的关键是了解 SpriteRenderer 组件。图 5-7 中卡片背面对象只有两个组件，场景中所有对象都有的标准 Transform 组件和一个新的 SpriteRenderer 组件。SpriteRenderer 组件使卡片背面成为精灵对象，并决定显示哪个精灵资源。注意，组件中的第一个属性为 Sprite，它链接到 Project 视图中的一个精灵。可以在代码中操作这个属性，而这也是上面的脚本所做的操作。

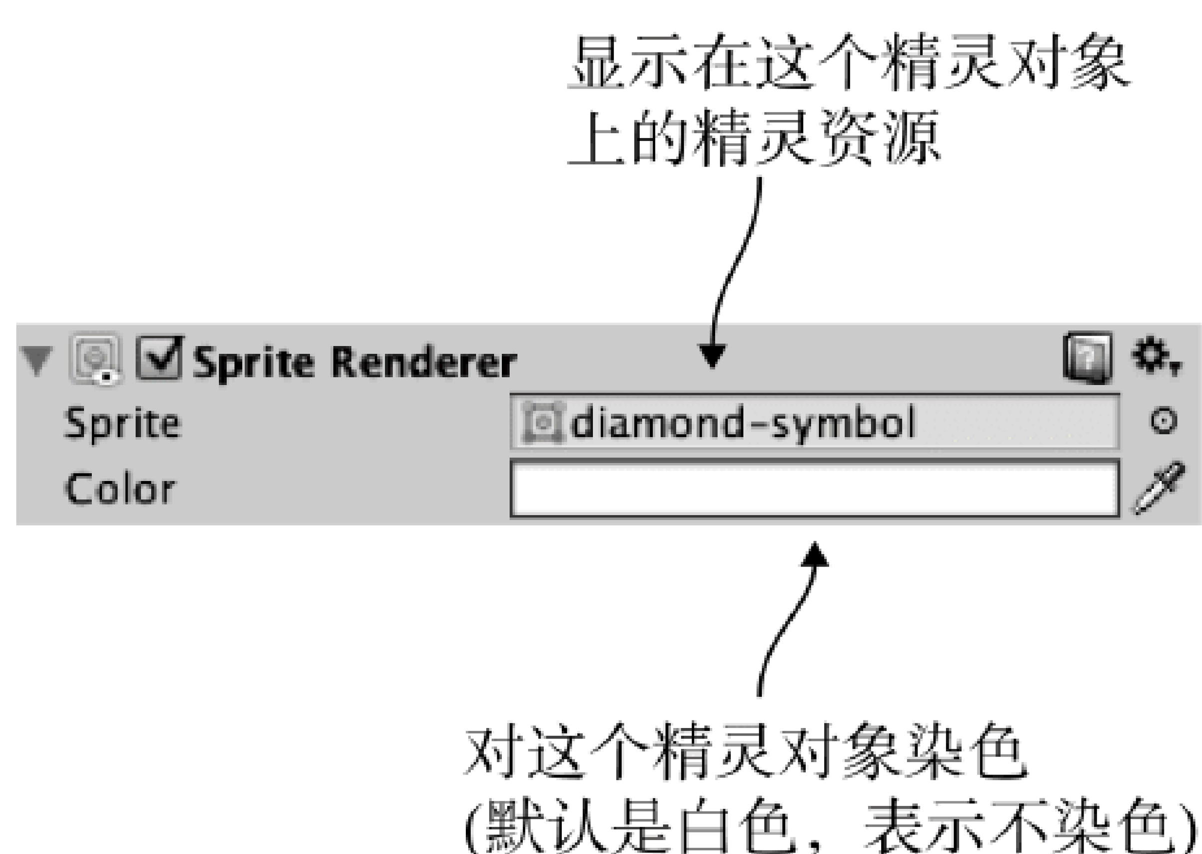


图 5-7　场景中的精灵对象上附有 SpriteRenderer 组件

如前面章节中自定义的脚本和 CharacterController 所示，GetComponent()方法返回同一对象上的其他组件，因此使用它获得 SpriteRenderer 对象的引用。SpriteRenderer 中的 sprite 属性可以设置为任何精灵资源，因此这段代码将该属性设置为顶部声明的 Sprite 变量(在编辑器中使用精灵资源填充的那个变量)。

这些工作不是很难！但仅涉及一个图像。我们需要使用 4 种不同的图像，因此现在删除代码清单 5.3 中的新代码(它只是演示了如何使用这项技术)，为下一节做准备。

5.3.2　通过不可见的 SceneController 设置图像

第 3 章在场景中创建一个不可见对象来控制对象的产生。这里也采用该方法，使用一个不可见对象来控制未关联到场景中任何特定对象的更抽象特性。首先创建一个空的 GameObject(记住，选择菜单 GameObject | Create Empty)。接着在 Project 视图中创建新的脚本 SceneController.cs，并将这个脚本资源拖动到控制器 GameObject 上。在新脚本中编写代码之前，首先将代码清单 5.4 中的内容添加到 MemoryCard 脚本，替代代码清单 5.3 中的代码。

代码清单 5.4　MemoryCard.cs 中的新公有方法

```
...
[SerializedField] private SceneController controller;

private int _id;
public int id {
    get {return _id;}
}

public void SetCard(int id, Sprite image) {
    _id = id;
    GetComponent<SpriteRenderer>().sprite = image;
}
...
```

添加 getter 函数(在 C#和 Java 等语言中通用的习惯叫法)

由于是公有方法，因此其他脚本能将新精灵传递到这个对象

SpriteRenderer 代码行和代码清单 5.3 中删掉的示例代码一样

前面代码清单中主要的改变是现在通过 SetCard()方法而不是 Start()方法设置精灵图像。由于 SetCard()是一个使用精灵作为参数的公有方法，因此可以从其他脚本中调用这个方法，并设置这个对象上的图像。注意，SetCard()还接受一个 ID 数字作为参数，代码会保存这个数字。尽管现在还不需要 ID，但接下来会编写匹配卡片的代码，而比较卡片是否匹配将依赖卡片的 ID。

注意　根据过去使用的编程语言，读者可能对“getter”和“setter”概念不熟悉。长话短说，这两个方法用于访问它们关联的属性(例如检索 card.id 的值)。使用 getter 和 setter 的原因很多，这个例子中 id 属性是只读的，因此它只提供 getter 方法而不提供 setter 方法。

最后，注意代码为控制器包含一个变量；即使 SceneController 开始克隆卡片对象来填充场景，卡片对象也需要引用控制器来调用它的公有方法。同往常一样，当代码引用场景中的对象时，只需要将 Unity 编辑器中的控制器对象拖动到 Inspector 的变量槽上即可。为单张卡片执行一次这个操作，之后所有复制的卡片对象都会拥有控制器的引用。

在 MemoryCard 中新增代码后，在 SceneController 中输入代码清单 5.5 的代码。

代码清单 5.5　首次处理记忆力游戏中的 SceneController

```
using UnityEngine;
using System.Collections;

public class SceneController : MonoBehaviour {
    [SerializeField] private MemoryCard originalCard;
    [SerializeField] private Sprite[] images;

    void Start() {
        int id = Random.Range(0, images.Length);
        originalCard.SetCard(id, images[id]);
```

用于引用场景中的卡片

引用精灵资源的数组

调用添加到 MemoryCard 中的公有方法

```
    }
}
```

现在这段短代码演示通过 SceneController 处理卡片的概念。大部分操作都很熟悉(例如，在 Unity 的编辑器中将卡片对象拖动到 Inspector 的变量槽上)，但图像数组是新概念。如图 5-8 所示，在 Inspector 中可以设置元素的个数。设置数组长度(也就是 Inspector 中的 size 属性)为 4，然后将用于卡片正面图像的精灵拖动到数组槽中。现在能通过数组访问这些精灵，就像其他对象引用一样。

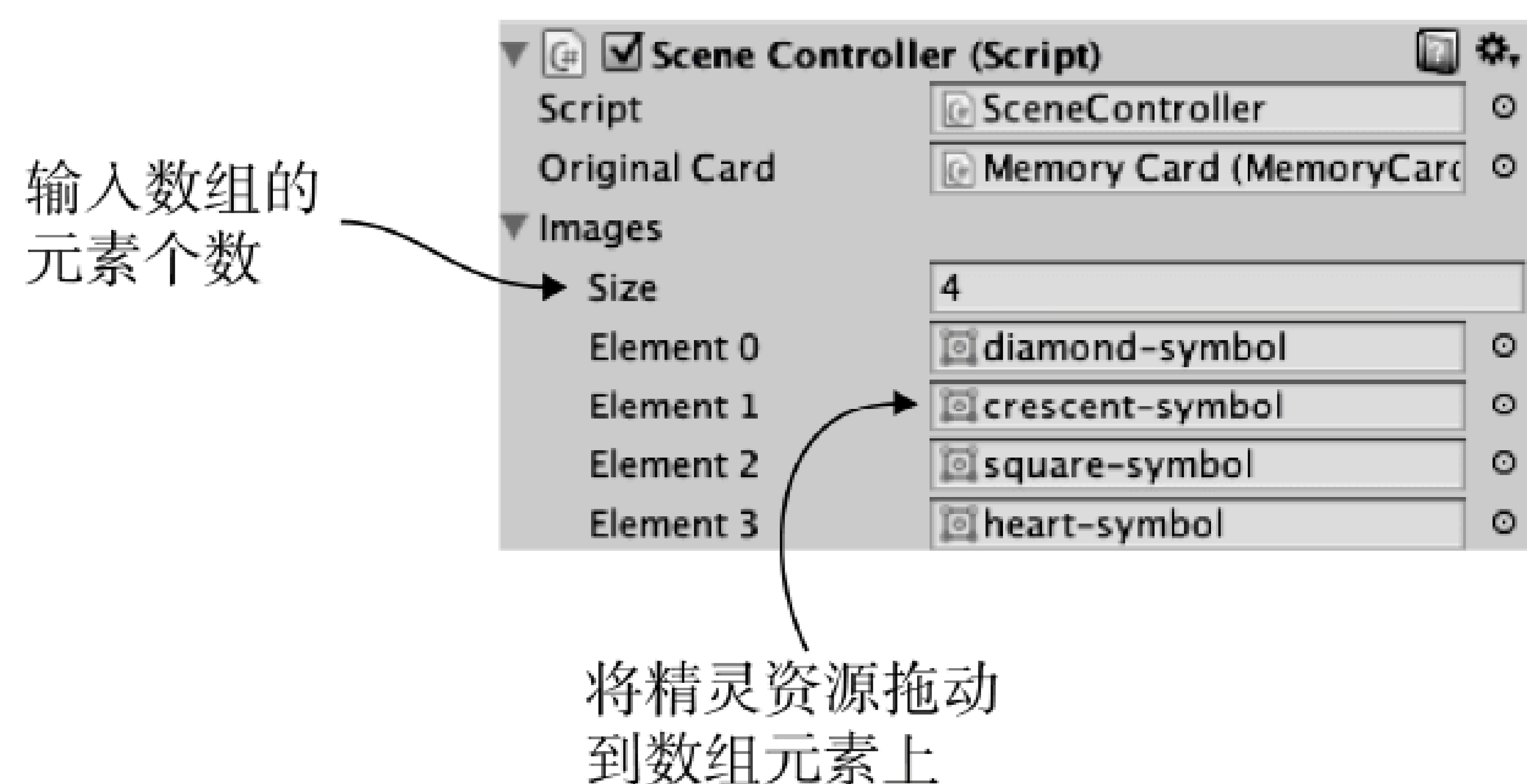


图 5-8　填充了精灵的数组

第 3 章使用了 Random.Range()方法，但那时并不关注它的精确边界值，但这次注意，其边界的最小值是包含的，且可能返回该最小值，但返回值总是小于最大值。

单击 Play 按钮，运行这段新代码。每次运行场景时，卡片正面都应用了不同的图像。下一步是创建所有卡片，而不是单张卡片。

5.3.3　实例化一叠卡片

SceneController 已经引用了卡片对象，因此现在使用 Instantiate()方法(参见代码清单 5.6)来克隆对象无数次，如第 3 章中产生对象一样。

代码清单 5.6　卡片克隆八次并定位到一个网格中

```
using UnityEngine;
using System.Collections;

public class SceneController : MonoBehaviour {
  public const int gridRows = 2;
  public const int gridCols = 4;
  public const float offsetX = 2;
  public const float offsetY = 2.5f;

  [SerializedField] private MemoryCard originalCard;
  [SerializedField] private Sprite[] images;

void Start() {
```

用于表示产生多少网格空间，以及它们之间距离的值

```
            Vector3 startPos = originalCard.transform.position;

            for (int i = 0; i < girdCols; i++) {
              for (int j = 0; j < gridRows; j++) {
                MemoryCard card;
                if ( i == 0 && j == 0 ) {
                 card = originalCard;
                } else {
                 card = Instantiate(originalCard) as MemoryCard;
                }

                int id = Random.Range(0, images.Length);
                card.SetGrid(id, images[id]);

                float posX = (offset * i) + startPos.x;
                float posY = - (offset * j) + startPos.y;
                card.transform.position = new Vector3(posX, posY, startPos.z);
              }
            }
          }
        }
```

第一张卡片的位置；所有其他卡片将从这里开始偏移

嵌套循环来定义网格的列和行

用于引用原始卡片或卡片副本的容器

对于 2D 图形，只需要偏移 X 和 Y，Z 坐标不变

尽管这段脚本比代码清单 5.5 长得多，但并没有太多地方需要解释，因为大多数新增的代码都是简单的变量声明和数学。这段代码中最古怪的地方可能是以 if(i==0 && j==0)开始的 if/else 语句。这个条件判断是选择原始卡片对象作为第一网格中的卡片，还是选择克隆卡片对象作为其他网格中的卡片。由于原始卡片已经存在于场景中，因此如果每次在循环迭代时都复制一个卡片对象，最后在场景中就会有过多的卡片对象。接着卡片根据循环迭代的次数对位置进行偏移。

提示　像移动 3D 对象一样，2D 对象能通过操作 transform.position 移动到屏幕上的不同位置，这个位置也可以在 Update()中重复递增。但当使用 transform.position 直接移动第一人称射击游戏的玩家时，碰撞检测将不起作用。为了在移动 2D 对象时碰撞检测仍旧有效，可以在赋予 Physics2D 组件之后调整 rigidbody2D.velocity。

现在运行代码，将会创建八格卡片(如图 5-9 所示)。准备这些卡片的最后一步是把它们组织成对，而不是完全随机。

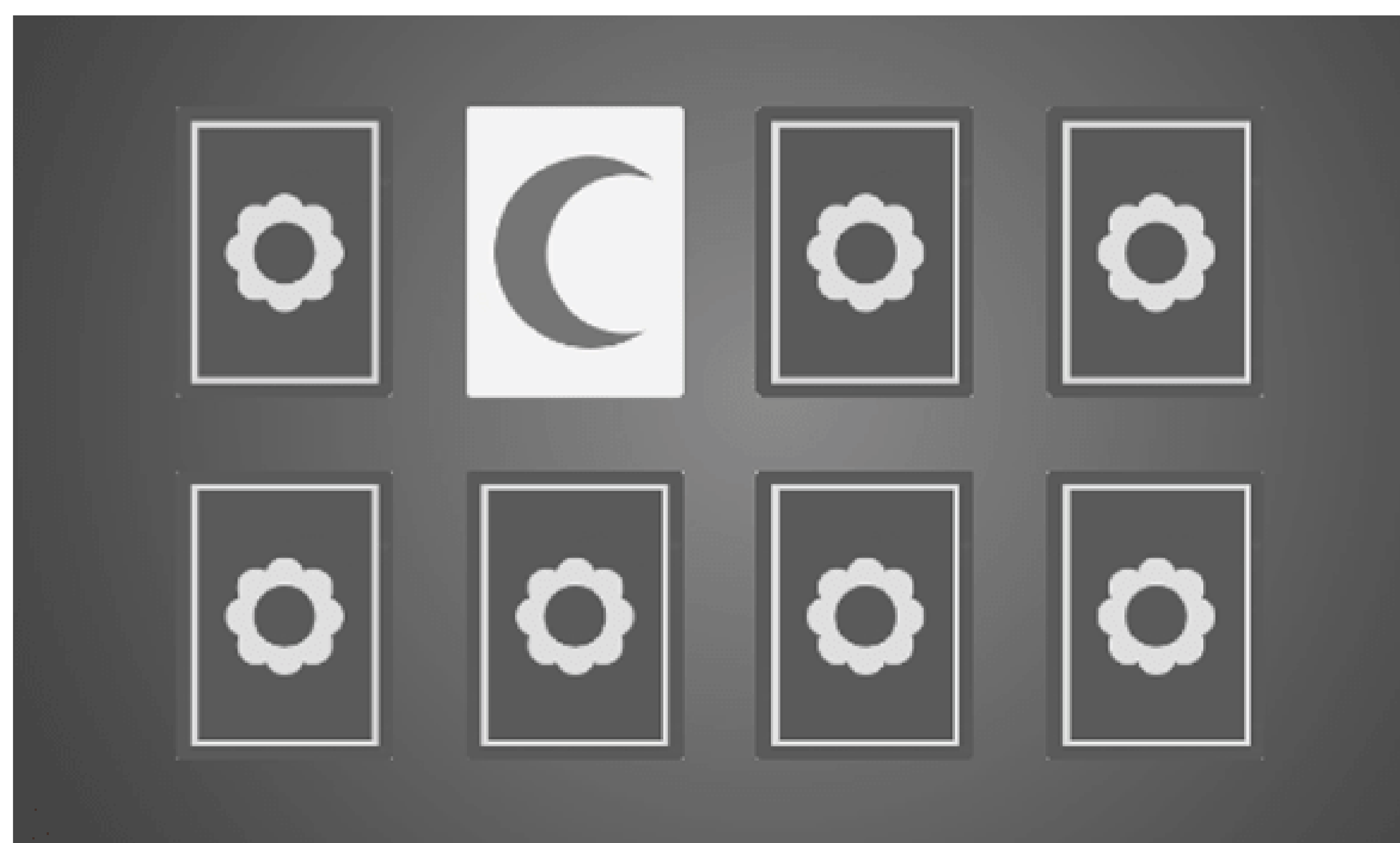


图 5-9　八格卡片，当单击它们时显示正面

5.3.4　打乱卡片

这里不是随机创建每张卡片，而是定义一个包含所有卡片 ID 的数组(0～3，每个数字出现两次，用于实现每对卡片)，然后打乱这个数组。接着在设置卡片时使用这个卡片 ID 数组，而不是随机生成每个卡片 ID。代码清单 5.7 展示了该代码。

代码清单 5.7　从打乱的列表中放置卡片

```
...
void Start() {                          ← 这个代码清单中的大部分代码是新增代码的上下文
  Vector3 startPos = originalCard.transform.position;

  int[] numbers = {0, 0, 1, 1, 2, 2, 3, 3};    ← 使用 ID 对为所有四种卡片精灵声明一个整型数组
  numbers = ShuffleArray(numbers);             ← 调用一个函数，打乱数组元素的顺序

  for (int i = 0; i < girdCols; i++) {
    for (int j = 0; j < gridRows; j++) {
      MemoryCard card;
      if ( i == 0 && j == 0 ) {
        card = originalCard;
      } else {
        card = Instantiate(originalCard) as MemoryCard;
      }

      int index = j * gridCols + i;
      int id = numbers[index];                 ← 从打乱的列表中取出 ID 而不是随机生成
      card.SetGrid(id, images[id]);

      float posX = (offsetX * i) + startPos.x;
      float posY = - (offsetY * j) + startPos.y;
      card.transform.position = new Vector3(posX, posY, startPos.z);
    }
  }
}

private int[] ShuffleArray(int[] numbers) {    ← 这是 Knuth 重排算法的实现
  int[] newArray = numbers.Clone() as int[];
  for (int i = 0; i < newArray.Length; i++ ) {
    int tmp = newArray[i];
    int r = Random.Range(i, newArray.Length);
    newArray[i] = newArray[r];
    newArray[r] = tmp;
  }
  return newArray;
}
...
```

现在当单击 Play 时，卡片会被打乱，并且每种卡片图像都会有两张。卡片数组由 Knuth(也称为费歇尔算法)重排算法运行，这是打乱数组元素的一种简单、有效的方式。该算法在数组循环中随机交换每个元素的位置。

可以单击所有的卡片来翻转它们，但记忆力游戏只支持处理一对，因此还需要更

多的代码。

5.4　实现匹配和匹配得分

完成功能完整的记忆力游戏的最后一步是检查匹配。尽管现在有一些格子卡片在单击时显示正面，但不同的卡片还不能互相影响。在记忆力游戏中，每次翻开一对卡片时，就需要检查翻开的卡片是否配对。

这个抽象逻辑——检查是否匹配并做出响应——需要在卡片被单击时通知 SceneController。这样就需要在 SceneController.cs 中添加代码清单 5.8 中的代码。

代码清单 5.8　SceneController 必须记录翻开的卡片

```
...
private MemoryCard _firstRevealed;
private MemoryCard _secondRevealed;

public bool canReveal {
   get {return _secondRevealed == null;}     // 当已经存在第二张翻开的卡片时，getter 方法返回 false
}
...
public void CardRevealed(MemoryCard card) {
   //initially empty
}
...
```

CardRevealed()方法会随时被填充，现在需要 CardRevealed()方法为空，以便能在 MemoryCard.cs 中访问它，且不产生任何编译错误。注意有一个只读的 getter 方法，这次 getter 方法用于判断是否翻开了另一张卡片，玩家在翻开的卡片未达到两张时才能翻开另一张其他卡片。

也需要修改 MemoryCard.cs，让卡片在被单击时调用当前为空的 SceneController 方法。根据代码清单 5.9 修改 MemoryCard.cs 中的代码。

代码清单 5.9　修改 MemoryCard.cs，翻开卡片

```
...
public void OnMouseDown() {
  if (cardBack.activeSelf && controller.canReveal) {     // 检查控制器的 canReveal 属性，确保一次只翻开两张卡片
     cardBack.SetActive(false);
     controller.cardRevealed(this);     // 当翻开卡片时通知控制器
  }
}

public void Unreveal() {     // 一个公有方法，因此 SceneController 可以再次隐藏卡片(通过重新显示 card_back)
  cardBack.SetActive(true);
```

```
   }
...
```

如果在 CardRevealed()中放置一条调试语句来测试对象间的通信，则只要单击卡片，就会显示测试消息。接下来先处理一张翻开的卡片。

5.4.1　保存并比较翻开的卡片

卡片对象被传入 CardRevealed()中，接下来开始跟踪翻开的卡片。编写代码清单 5.10 中的代码。

代码清单 5.10　在 SceneController 中跟踪翻开的卡片

```
...
public void CardRevealed(MemoryCard card) {
   if (_firstRevealed == null) {
      _firstRevealed = card;
   } else {
      _secondRevealed = card;
      Debug.Log("Match? "+(_firstRevealed.id == _secondRevealed.id));
   }
}
...
```

将卡片对象存储在两个卡片变量中的一个，这取决于第一个变量是否已被占用

比较两个翻开卡片的 ID

代码清单将翻开的卡片保存在两个卡片变量中的一个，这取决于第一个变量是否已被占用。如果第一个变量为空，则将翻开的卡片赋予它；如果它已被占用，则把翻开的卡片赋予第二个变量，并检查 ID 是否匹配。调试语句在控制台中输出 true 或 false。

现在代码还不能响应匹配——它只是检查它们。接下来编写代码响应匹配。

5.4.2　隐藏不匹配的卡片

下面将再次使用协程(coroutine)，因为对不匹配卡片的响应是暂停一下，允许玩家查看卡片。有关协程的解释可以参阅第 3 章。长话短说，使用协程允许在检查是否匹配前暂停响应。代码清单 5.11 列出了添加到 SceneController 中的代码。

代码清单 5.11　SceneController，匹配得分或隐藏错误匹配

```
...
private int _score = 0;
...
public void CardRevealed(MemoryCard card) {
   if (_firstRevealed == null) {
     _firstRevealed = card;
   } else {
      _secondRevealed = card;
      StartCoroutine(CheckMatch());
   }
}
```

添加到 SceneController 顶部附近的列表中

这个函数中唯一改变的一行，当两张卡片都翻开时调用协程

```
private IEnumerator CheckMatch() {
  if (_firstRevealed.id == _secondRevealed.id) {
    _score++;
    Debug.Log("Score: " + _score);
  }
  else {
    yield return new WaitForSeconds(.5f);

    _firstRevealed.Unreveal();
    _secondRevealed.Unreveal();
  }

    _firstRevealed = null;
    _secondRevealed = null;
}
...
```

如果翻转的卡片有匹配的 ID，则增加分数

如果卡片不匹配，则隐藏卡片

不管是否匹配，都清除变量的内容

首先添加一个_score 值用于跟踪分数，接着当第二张卡片翻开时运行协程 CheckMatch()。在该协程中有两个代码执行路径，这取决于卡片是否匹配。如果卡片匹配，则协程不会暂停，yield 命令会被跳过。但如果卡片不匹配，协程则在调用两个卡片的 Unreveal()之前暂停 0.5 秒，再次隐藏卡片。最后，不管卡片是否匹配，用于存储卡片的变量都会设置为 null，为翻开更多的卡片做准备。

运行游戏时，不匹配的卡片会在隐藏前短暂显示。当匹配加分时会显示调试消息，但分数需要显示在屏幕的标签中。

5.4.3　显示分数的文本

将信息显示给玩家是游戏中存在 UI 的一半原因(另一半原因是接收玩家的输入，UI 按钮的内容在下一节中讨论)。

定义　和 UI(User Interface，用户界面)紧密相关的术语是 GUI(Graphical User Interface，图形用户界面)，指的是接口中的可视化部分，例如文本、按钮，许多人把这些部分都称为 UI。

Unity 有多种创建文本显示的方式。一种方式是在场景中创建 3D 文本对象。这是一个特殊的网格组件，首先要创建一个空对象，并将这个组件附加到空对象上。从 GameObject 菜单选择 Create Empty，单击 Add Component 按钮，并选择 Mesh | Text Mesh。

注意　3D 文本听起来和 2D 游戏不兼容，但是不要忘记这只是技术上的 3D 场景，它看起来是平坦的，因为这个场景是通过一个正交摄像机来观察的。这意味着可以在需要时将任何 3D 对象放到 2D 游戏中——它们只是以平面透视的方式显示。

提示 这个解释使用了 Unity 标准的 Text Mesh 组件，而自己的项目应该考虑使用 TextMesh Pro。这是一个在外部开发的、改进的文本系统，最近才整合到 Unity 中。

把这个对象定位在(-4.75, 3.65, -10)，也就是向左 475 像素，向上 365 像素，把它放到左上角，并且更靠近摄像机，使它显示在其他游戏对象的上方。在 Inspector 中，找到底部附近的 Font 设置，单击小圆圈按钮，打开文件选择器，接着选择可见的 Arial 字体。输入"Score:"作为 Text 设置。正确的定位也需要将 Anchor 设置为 Upper left(这可以控制文字在输入后的扩展方式)，因此如果有需要就修改它。默认情况下，文本会显得模糊，但调整如图 5-10 所示的设置，就可以轻松地修复此问题。

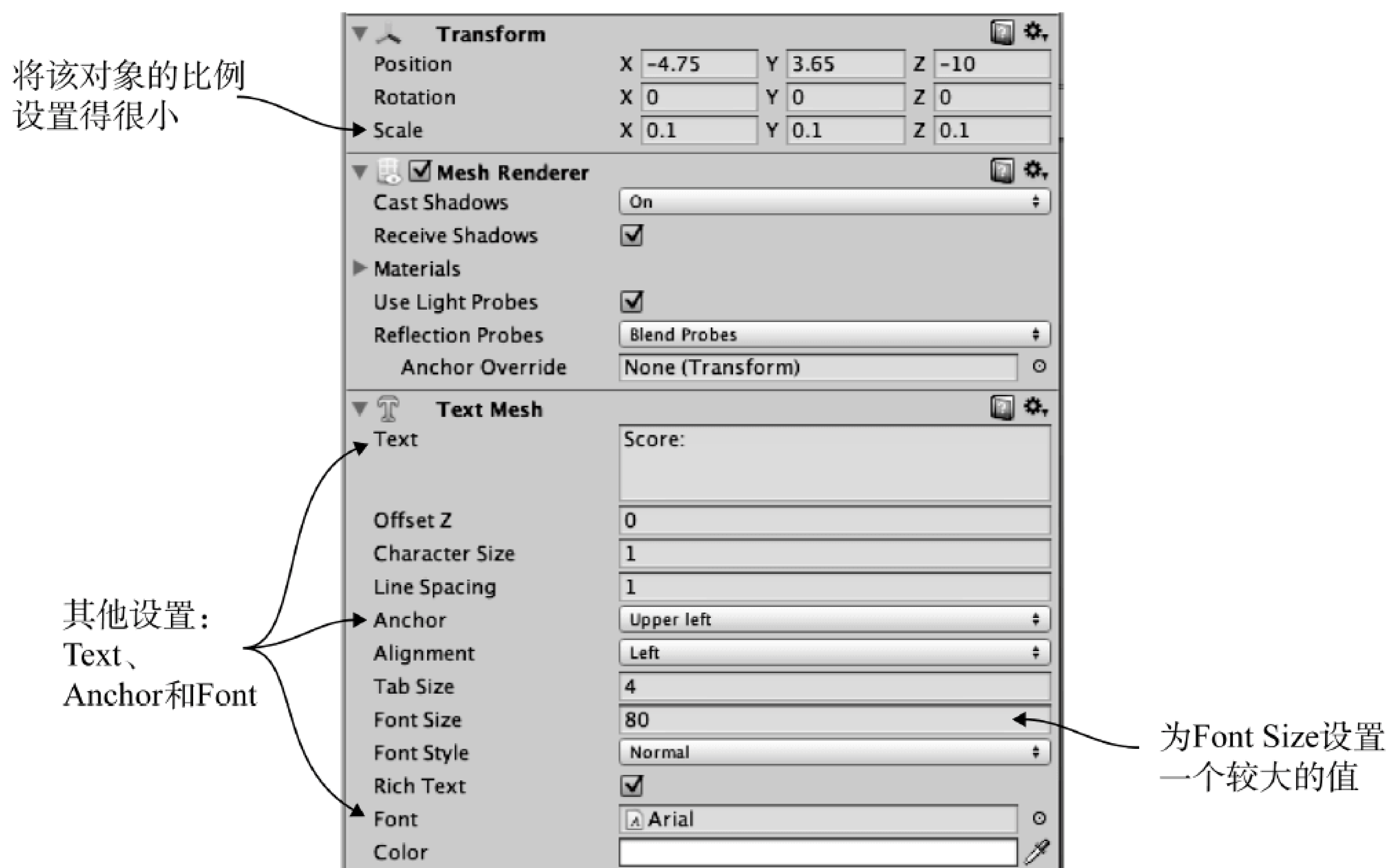


图 5-10　用于文本对象的 Inspector 设置，可以让文本更清晰

如果将新的 TrueType 字体导入到项目中，就可以采用新字体，但对这里的目的而言，默认字体也很好。奇怪的是，需要做一些大小调整，才能让默认文本看起来更清晰。首先设置 TextMesh 组件的 Font Size 为一个很大的值(使用 80)。现在将对象缩放到很小，如(0.1, 0.1, 1)。增加 Font Size 为文本显示增加了很多像素，缩放对象把那些像素压缩到一个更小的空间。

要处理这些文本对象，只需要在分数代码中做一些调整(查看代码清单 5.12)。

代码清单 5.12　在文本对象上显示分数

```
...
[SerializeField] private TextMesh scoreLabel;
...
private IEnumerator CheckMatch() {
   if (_firstRevealed.id == _secondRevealed.id) {
      _score++;
```

```
        scoreLabel.text = "Score: " + _score;
    }
...
```

显示的文本是在文本对象上设置的属性

可以看出，text 是对象的一个属性，可以将它设置为一个新字符串。将场景中的文本拖动到刚刚添加到 SceneController 的那个变量上，接着单击 Play。现在运行游戏并单击匹配的卡片时，应该能看到显示的分数。现在，游戏可以正常工作了！

5.5 重启按钮

此时，记忆力游戏已经具备了一些功能。可以运行该游戏，所有必要的特性都已经具备。但这个核心游戏还缺少玩家在已完成的游戏中需要的首要功能。例如，现在只能运行一次游戏；需要退出并重启，才能重新运行游戏。接下来给屏幕添加一个控件，让玩家能在不退出游戏的情况下就能重新运行游戏。

这个功能拆分为两个任务：创建 UI 按钮和当单击按钮时重置游戏。图 5-11 显示了带 Start 按钮的游戏的外观。

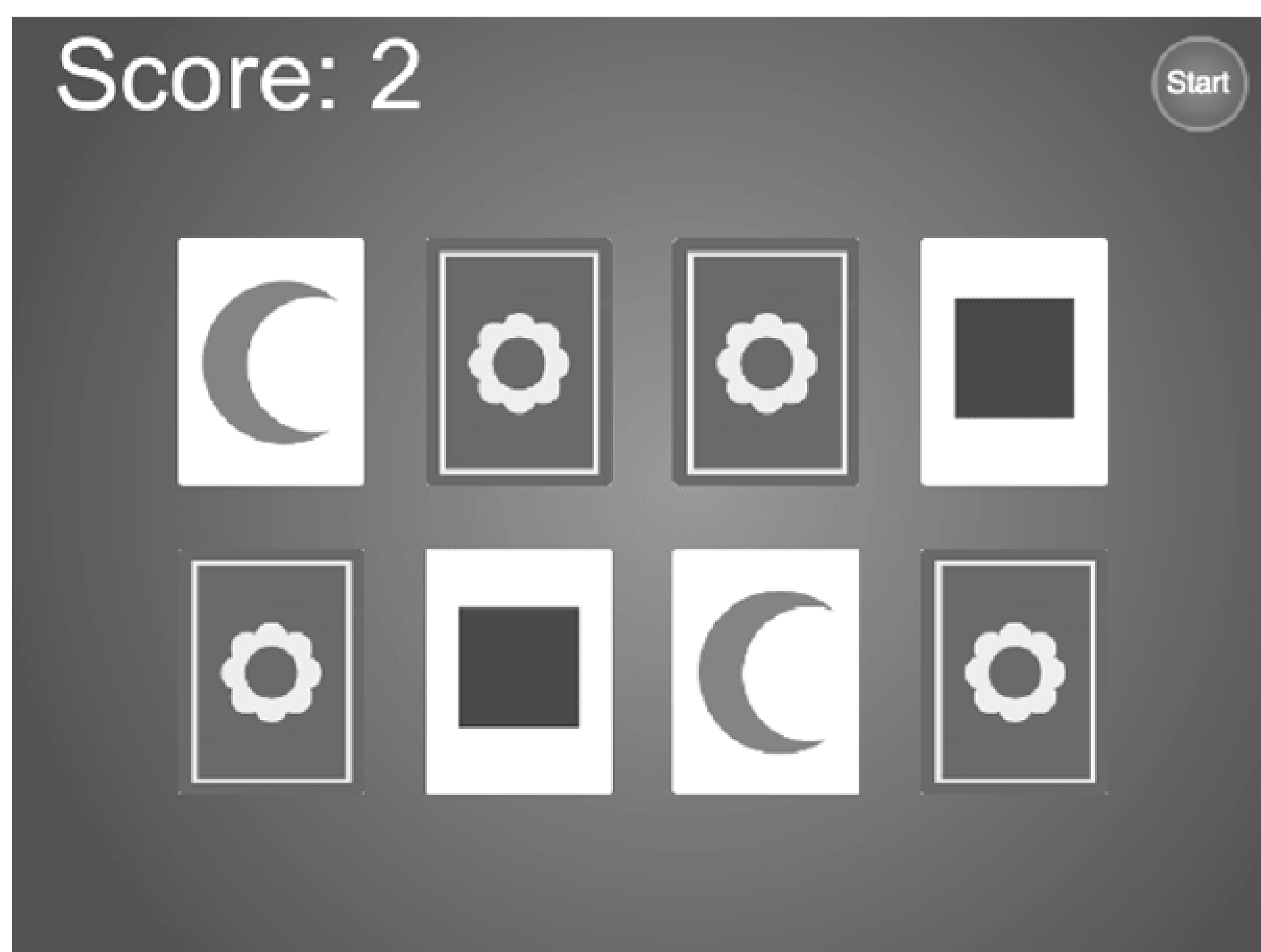


图 5-11　完整的记忆力游戏的屏幕，包含 Start 按钮

这两个任务都不是 2D 游戏特有的，所有游戏都需要 UI 按钮，所有游戏需要能重置。接下来介绍这两个任务以结束本章的讨论。

5.5.1 使用 SendMessage 编写 UIButton 组件

首先在场景中放置按钮精灵，从 Project 视图把它拖到场景中。设置位置，诸如 (4.5, 3.25, −10)，这会把按钮放在右上角(中心往右 450 像素，往上 325 像素)，移动它使它更接近摄像机，以便出现在其他游戏对象的顶部。因为我们希望这个对象可以单击，所以为它添加一个碰撞器(就像对卡片对象的处理一样，选择 Add Component | Physics 2D | Box Collider)。

注意 Unit 为创建 UI 显示提供了多种方式，包括最新版 Unity 中的高级 UI 系统。现在使用标准的显示对象来构建一个按钮。第 6 章将介绍高级 UI 系统的功能，2D 和 3D 游戏中的 UI 也都可以使用该系统构建。

现在创建一个称为 UIButton.cs 的新脚本(如代码清单 5.13 所示)，并把这个脚本赋给按钮对象。

代码清单 5.13 创建通用且可重用的 UI 按钮

```
using UnityEngine;
using System.Collections;

public class UIButton : MonoBehaviour {
  [SerializeField] private GameObject targetObject;
  [SerializeField] private string targetMessage;
  public Color highlightColor = Color.cyan;

  public void OnMouseEnter() {
    SpriteRenderer sprite = GetComponent<SpriteRenderer>();
    if (sprite != null) {
      sprite.color = highlightColor;
    }
  }
  public void OnMouseExit() {
    SpriteRenderer sprite = GetComponent<SpriteRenderer>();
    if (sprite != null) {
      sprite.color = Color.white;
    }
  }

  public void OnMouseDown() {
    transform.localScale = new Vector3(1.1f, 1.1f, 1.1f);
  }
  public void OnMouseUp() {
    transform.localScale = Vector3.one;
    if (targetObject != null) {
      targetObject.SendMessage(targetMessage);
    }
  }
}
```

引用一个用于通知单击的目标对象

当鼠标悬停在按钮上时对该按钮染色

当单击按钮时按钮稍微变大

当单击按钮时将消息发送到目标对象

这段代码主要由一系列 OnMouseSomething 函数构成。类似 Start()和 Update()，这些是 Unity 中对所有脚本组件都自动可用的函数。如果对象有一个碰撞器，这些函数就会响应鼠标交互。MouseEnter 和 MouseExit 是一对用于处理鼠标悬停在对象上和离开对象的事件。MouseEnter 在鼠标光标首次悬停在对象上时触发，MouseExit 在鼠标光标移开时触发。类似的，MouseDown 和 MouseUp 是一对用于处理单击鼠标的事件。MouseDown 在鼠标按钮被按下时触发，而 MouseUp 在鼠标按钮被释放时触发。

可以看到，当鼠标悬停时精灵被染色，而当单击时精灵被放大。当鼠标开始交互

时，可以观察到两种情况下的变化(颜色或大小)，当鼠标交互结束时，属性还原为默认值(白色或缩放为 1)。对于缩放，代码使用了所有 GameObject 都有的标准 transform 组件。而对于染色，代码使用了精灵对象拥有的 SpriteRenderer 组件，精灵被设置为一个颜色，该颜色通过公有变量在 Unity 的编辑器中定义。

除了让比例返回到 1，在释放鼠标时还调用了 SendMessage()。SendMessage()调用 GameObject 的所有组件中指定方法名的方法。这里消息的目标对象和要发送的消息都由序列化变量定义。通过这种方式，在 Inspector 中把不同按钮的目标设置为不同的对象，相同的 UIButton 组件就可以用于各种按钮。

通常，在 C#这类强类型语言中使用面向对象编程时，需要知道目标对象的类型才能和该对象通信(例如，为了调用该对象的公有方法，需要 targetObject.SendMessage())。但 UI 元素的脚本可能会有很多不同类型的目标，因此 Unity 提供了 SendMessage()方法，即使不知道对象的具体类型，也可以通过该方法把指定的消息通报给目标对象。

警告　对于 CPU 而言，使用 SendMessage()的效率比调用已知类型的公有方法低(也就是 object.SendMessage("Method")与 component.Method()相比)，因此只有当使用 SendMessage()能够让代码更易于理解和工作时，才比较好。一般规则是，仅当有很多不同类型的对象接收消息时，才需要使用 SendMessage()，在这种情况下，继承或接口的不灵活性甚至会阻碍游戏开发进程，体验也不好。

编写完这段代码后，在按钮的 Inspector 上把公有变量关联好。可以将高亮颜色设置为自己喜欢的颜色(但默认的蓝绿色在蓝色按钮上看起来很不错)。同时，将 SceneController 对象放在目标对象槽上，并输入 Restart 作为消息。

如果现在运行游戏，右上角就会出现 Reset 按钮，响应鼠标时会改变其颜色，而当单击按钮时，按钮看起来像轻微“弹出”。但现在单击按钮，会显示一个错误消息，在控制台中会显示一个错误，说明还没有 Restart 消息的接收器。这是因为我们还没有在 SceneController 中编写 Restart()方法，下一步将添加该方法。

5.5.2　从 SceneController 中调用 LoadScene

按钮的 SendMessage()尝试调用 SceneController 中的 Restart()方法，因此接下来添加这个方法(见代码清单 5.14)。

代码清单 5.14　SceneController 中重新加载关卡的代码

```
...
using UnityEngine.SceneManagement;        ←—— 包含 SceneManagement 代码
...
public void Restart() {
   SceneManager.LoadScene("Scene");       ←—— 使用这个命令加载场景资源
}
...
```

可以发现，Restart()调用了 LoadScene()。该命令加载了一个已保存的场景资源(即在 Unity 中单击 Save Scene 时创建的文件)。把要加载的场景名称传入 LoadScene()方法，在本例中场景保存为 Scene，如果使用了不同的场景名称，就将这个名称传入该方法。

单击 Play，观察发生了什么。翻开一些卡片并做一些匹配。如果单击 Reset 按钮，游戏将重新开始，所有卡片隐藏，而且分数为 0。很好，这正是我们想要的效果！

LoadScene()方法能加载不同场景。当加载场景时究竟发生了什么？为什么这样做能重置游戏？实际上，当加载不同关卡时，当前关卡中所有的内容(场景中所有对象以及对象上所有附加的脚本)都从内存中清除，接着从新场景中加载所有对象。由于本例中的“新”场景就是当前场景中所保存的资源，因此清除所有对象，之后重新加载它们。

提示　可以标记指定的对象，使它在加载关卡时不会从内存中被清除。Unity 提供了 DontDestroyOnLoad()方法来保证对象在多个场景中存在，后续章节将在代码架构上使用这个方法。

又成功完成了一个游戏！“完成”是一个相对的术语，还可以实现更多功能，但初始计划中的所有工作都已完成。这个 2D 游戏中的很多概念也能应用到 3D 游戏中，特别是有关游戏状态检查和加载关卡方面的技术。后面将退出这个记忆力游戏，去开始新的项目。

5.6　小结

- 使用正交摄像机在 Unity 中显示 2D 图形。
- 为了图形上像素完美，摄像机的大小应该为屏幕高度的一半。
- 单击精灵前首先需要为精灵添加 2D 碰撞器。
- 可以通过编程为精灵加载新图像。
- UI 文本可以使用 3D 文本对象建立。
- 加载关卡可以重置场景。

第 6 章

创建基本的2D 平台游戏

本章涵盖：

- 不断移动精灵
- 播放精灵表动画
- 2D 物理(碰撞、重力)
- 侧滚游戏的摄像机控制

下面创建一个新游戏，并继续学习 Unity 的 2D 功能。第 5 章介绍了基本概念，因此本章将在这些基础上创建一个更复杂的游戏。具体来说，要构建一个 2D 平台游戏的核心功能。这款游戏也称为平台游戏，是一款常见的 2D 动作游戏，以《超级马里奥兄弟》等经典游戏而闻名。从侧面观看，角色在平台上奔跑跳跃，视图滚动着跟随。图 6-1 显示了最终结果。

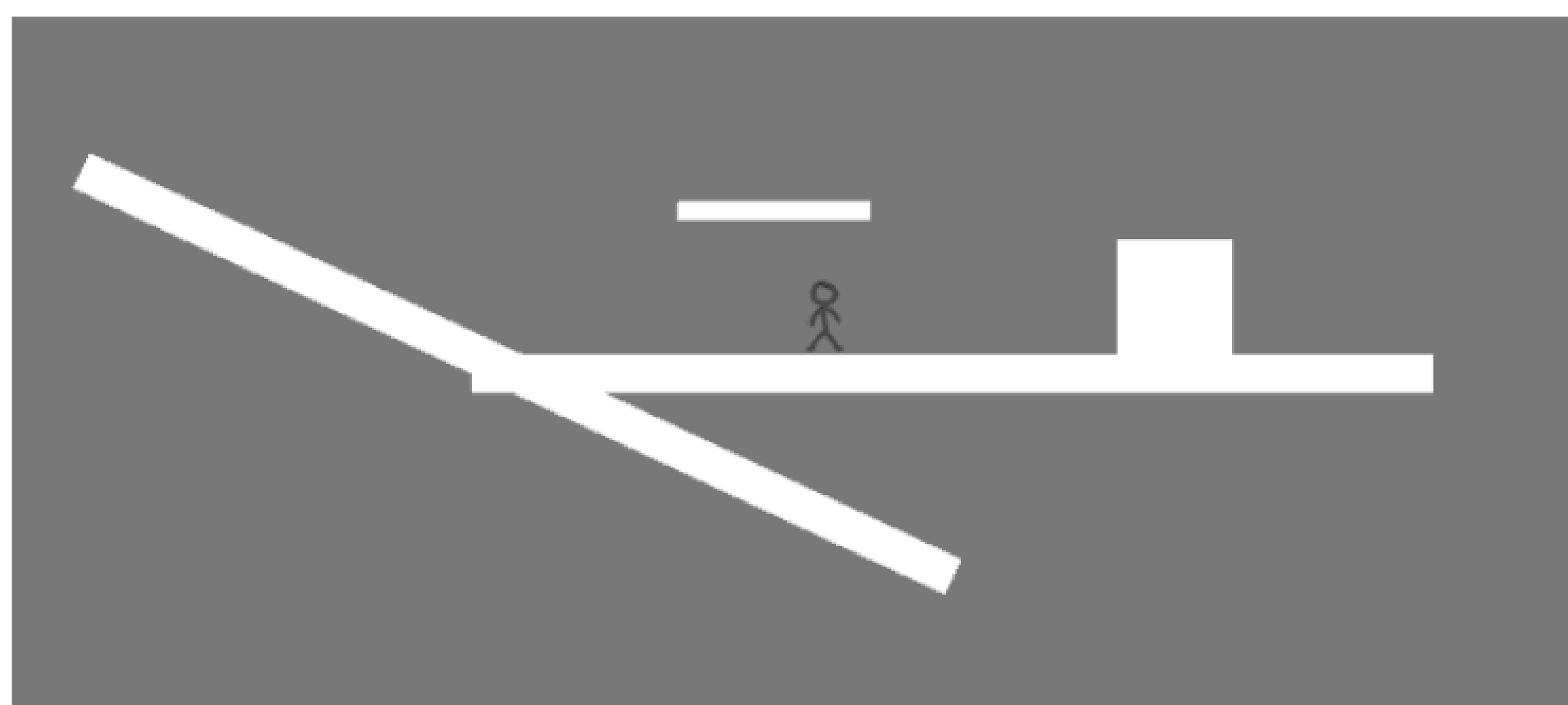


图 6-1　本章的最终产品

这个项目将介绍一些概念，比如左右移动玩家，播放精灵的动画，以及增加跳跃功能。还要介绍平台游戏中常见的几个特性，比如单向楼层和移动平台。从这个 shell 到一个完整的游戏，都要重复这些概念。

首先，在 2D 模式下创建一个新项目：在 Unity 的打开窗口中选择 New，或在 File 菜单下选择 New Project，然后在出现的窗口中选择 2D。在新项目中创建两个文件夹，称为 Sprites 和 Scripts，以组织各种资源。可以像上一章那样调整摄像机，但现在只需要将 Size 缩小到 4。这个项目不需要完美的摄像机设置，但需要给已经准备好发布的游戏调整大小。

提示 *屏幕中央的摄像机图标可能会碍事，所以可以利用 Gizmos 菜单隐藏它。Scene 视图的顶部是 Gizmos 的标签，单击按字母顺序排列的列表的标签，再单击 Camera 旁边的图标。*

现在保存空的场景(当然，在工作时要周期性地单击 Save)，以创建这个项目中的 Scene 资源。目前所有的东西都是空的，所以第一步是引入美术资源。

6.1 设置图形

在编写 2D 平台游戏的功能之前，需要将一些图像导入到项目中(记住，2D 游戏中的图像称为精灵，不称为贴图)，然后将这些精灵放到场景中。这款游戏是一个 2D 平台游戏的外壳，玩家控制的角色在一个几乎是空的基本场景中运行，所以只需要用于平台和玩家的两个精灵。下面分开介绍每个精灵，因为虽然本例中的图片很简单，但是其中有一些不明显的细节。

6.1.1 放置墙壁和地板

简单地说，这里需要使用一个空白的白色图像。本章示例项目中包含的图像是 blank.png。下载示例，并从中复制 blank.png。抓取 PNG 文件，将其拖放到新项目的 Sprite 文件夹中，并确保在 Inspector 中，Import Settings 指定它是 Sprite，而不是 Texture(对于 2D 项目应自动设置为 Sprite，但应反复检查)。

现在所做的操作基本上和第 4 章中的白盒是一样的，不过是 2D，而不是 3D。在 2D 世界中，白盒是用精灵而不是网格来完成的，但是保留了同样的活动，即为阻止玩家四处移动的空白地板和墙壁。

要放置地板对象，将空白精灵拖动到场景中，如图 6-2 所示，大约位置是(0.15, −1.27, 0)，缩放到(50, 2, 1)，并将其名称更改为 Floor。然后拖曳另一个空白精灵，缩放到(6, 6, 1)把它放在地板右边，大约位置是(2, −0.63, 0)，并将其命名为 Block。

这很简单，现在地板和积木都做好了。下一个需要的对象是玩家的角色。

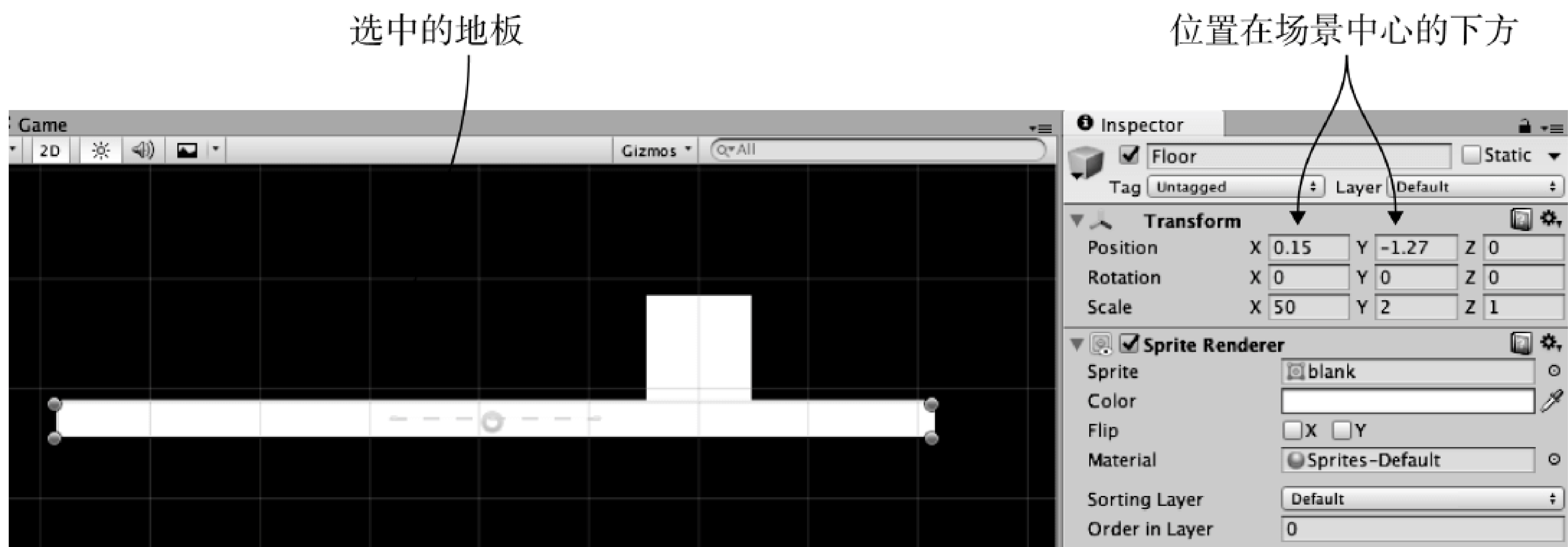


图 6-2　地板平台的位置

6.1.2　导入精灵表

唯一需要的美术资源是玩家的精灵，所以还要复制示例项目中的 stickman.png。但与空白图像不同的是，这次它是组合成一张图像的一系列单独精灵。如图 6-3 所示，stickman 图像是两个动画的帧：闲着站立和步行循环。我们不会详细讨论如何制作动画，但 idle 和 cycle 都是游戏开发者常用的术语。idle 指无所事事时的细微运动，而 cycle 则是持续循环的动画。

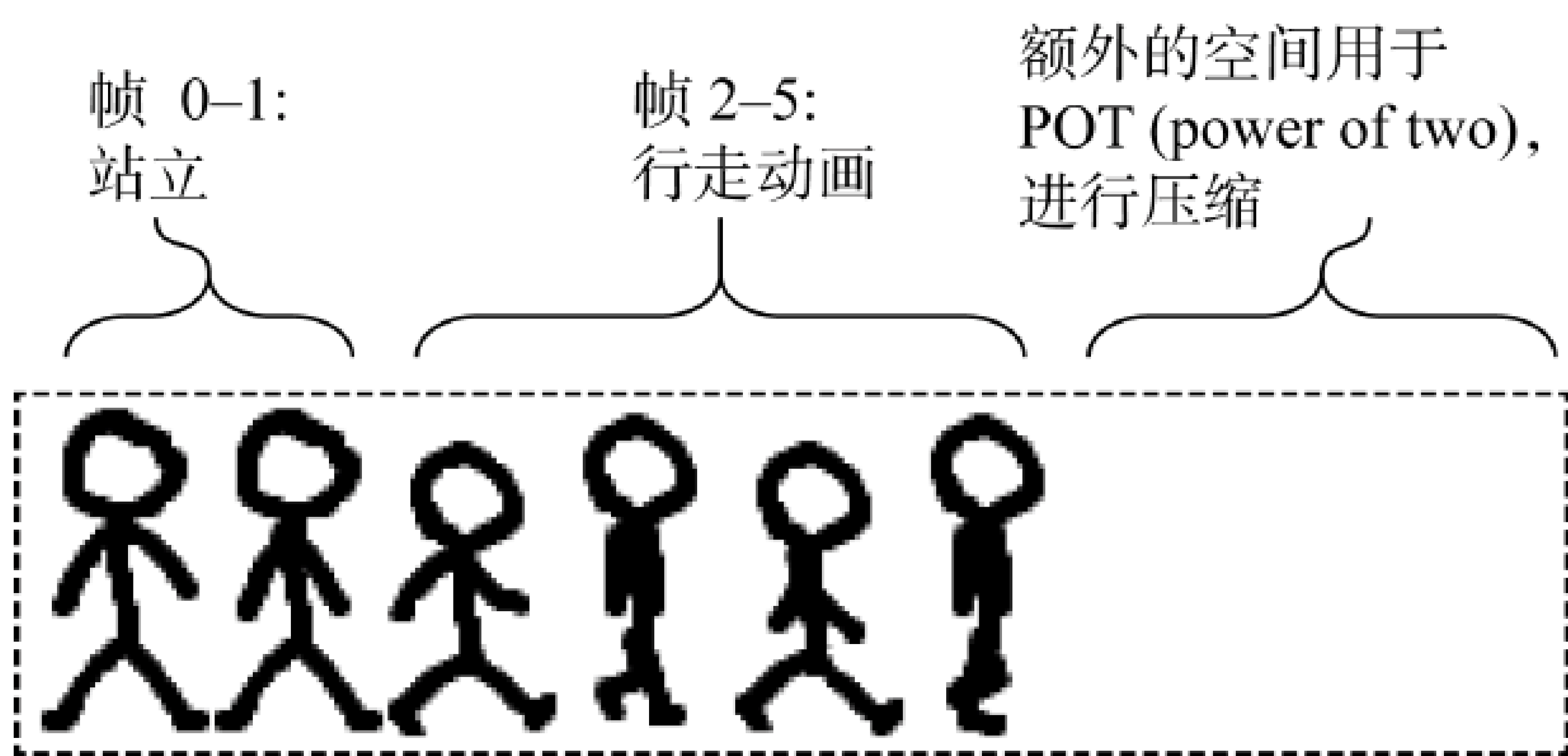


图 6-3　stickman 精灵表：一行 6 帧

如前一章所述，图像文件可能是一堆精灵图像组合在一起，而不仅仅是一个精灵。当多个精灵图像是动画的各个帧时，这样的图像称为精灵表。在 Unity 中，精灵表仍然会以单个资源的形式出现在 Project 视图中，但是如果单击资源上的箭头，它将展开，并显示所有单个精灵图像。图 6-4 显示了它的外观。

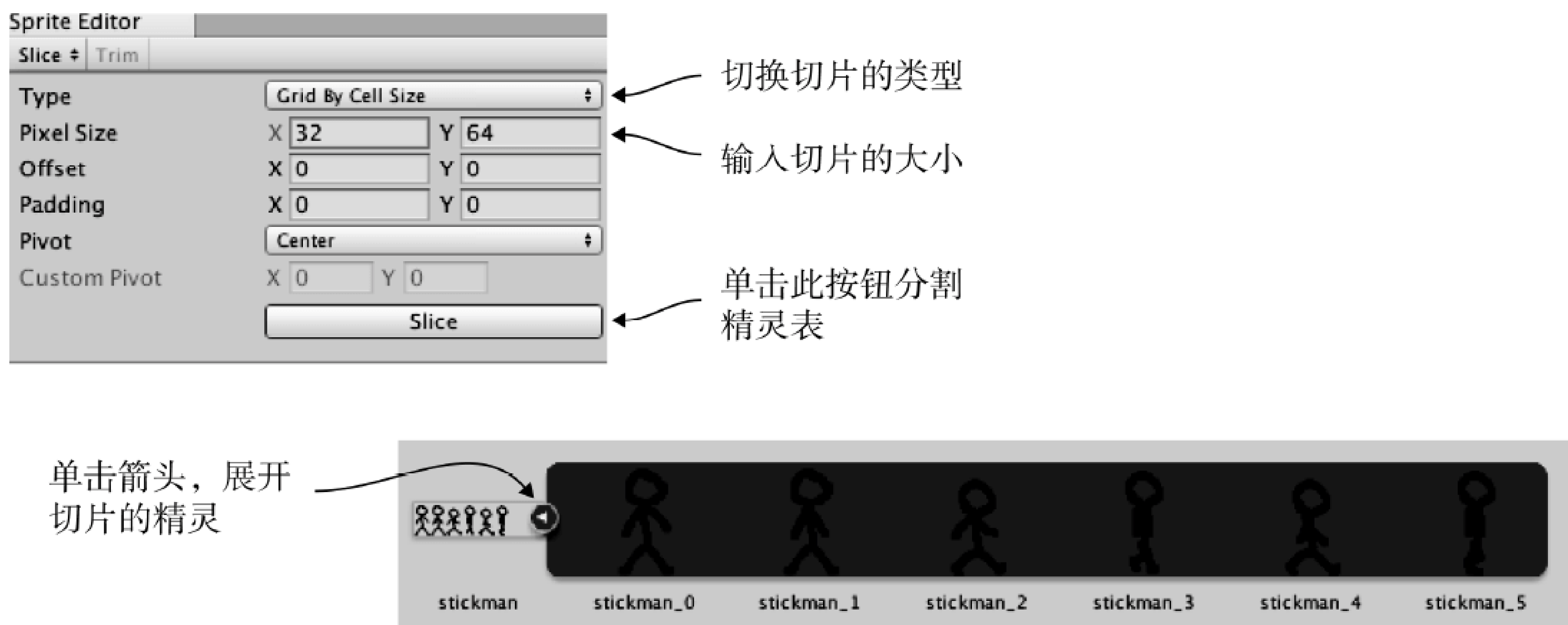


图 6-4　把精灵表切割为单独帧

把 stickman.png 拖入 Sprites 文件夹以导入图像，但这一次在 Inspector 中改变了很多导入设置。选择精灵资源，将 Sprite 模式设置为 Multiple，然后单击 Sprite Editor 打开该窗口。单击窗口左上角的 Slice，将 Type 设置为 Grid By Cell Size (如图 6-4 所示)，Size 使用(32, 64)(这是精灵表中每个帧的大小)，然后单击 Slice 以查看分割后的帧。现在关闭 Sprite Editor 窗口，并单击 Apply 以保存更改。

精灵资源现在被分割，所以单击箭头展开帧；将一个(可能是第一个)stickman 精灵拖到场景中，将它放在地板中间，并命名为 Player。现在，玩家对象在场景中!

6.2　左右移动玩家

现在图形已经设置好了，下面开始为玩家的移动编程。首先，场景中的玩家实体需要一些额外的组件供我们控制。如前几章所述，Unity 中的物理模拟作用于具有 Rigidbody 组件的对象时，我们希望将物理定律(特别是碰撞和重力)作用于该角色。同时，角色还需要一个 Collider 组件来定义其边界，以进行碰撞检测。这些组件之间的区别是微妙而重要的：Collider 定义了物理定律作用的形状，Rigidbody 指示物理模拟要作用的对象。

注意　这些组件是分开的(尽管它们是密切相关的)，因为许多不需要物理模拟的对象确实需要在物理定律的作用下与其他对象碰撞。

需要注意的另一个微妙之处是 Unity 为 2D 游戏提供了一个独立的物理系统，而不是 3D 物理。因此，本章使用 Physics 2D 部分的组件，而不是列表中的常规 Physics 部分。

在场景中选择 Player，然后在 Inspector 中单击 Add Component，如图 6-5 所示，在菜单中选择 Physics 2D | Rigidbody 2D。现在再次单击 Add Component，添加 Physics 2D | Box Collider 2D。Rigidbody 需要少量的微调，所以在 Inspector 中将 Collision Detection 设置为

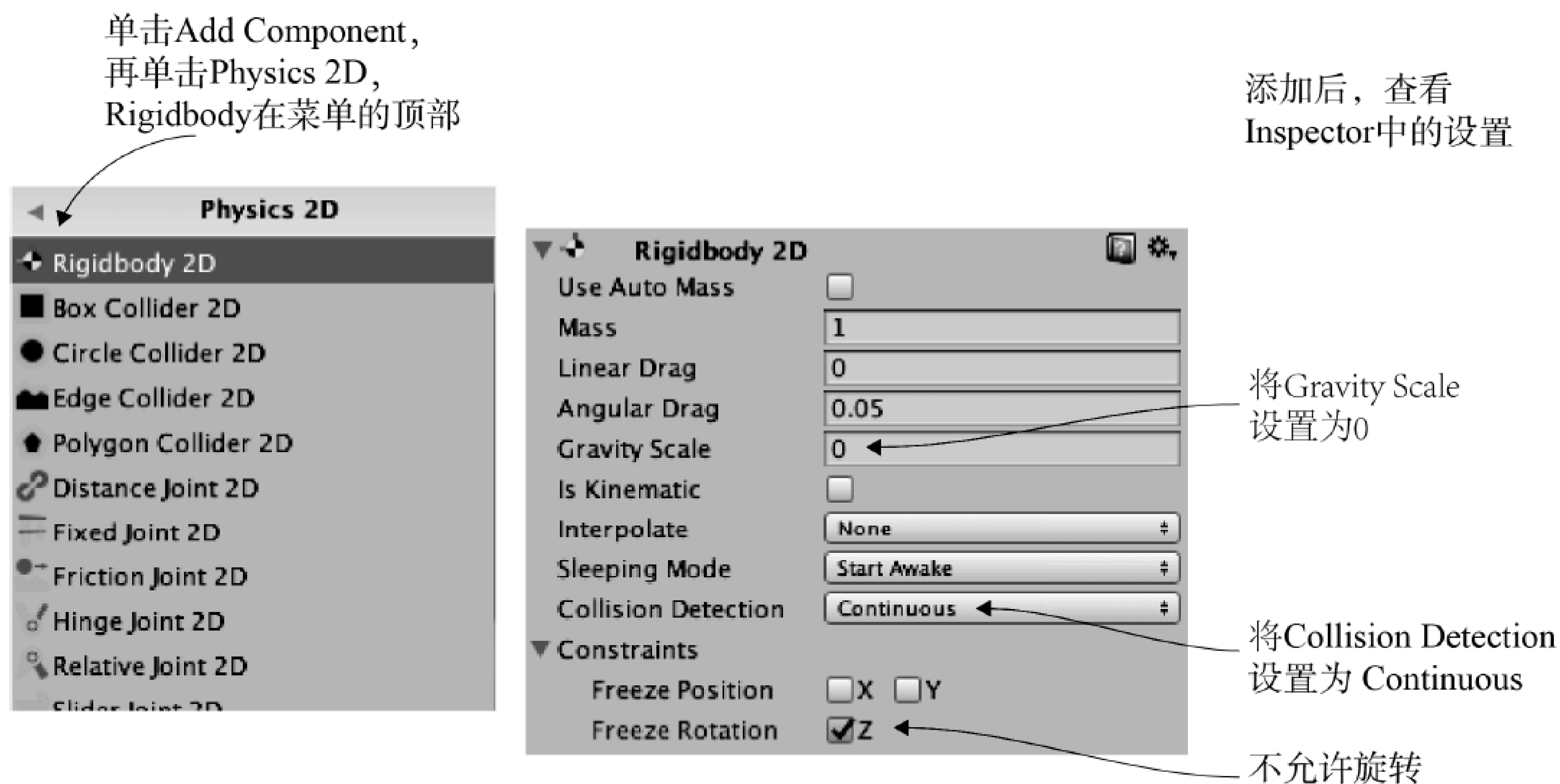


图 6-5　添加并调整 Rigidbody 2D 组件

Continuous, Freeze Rotation Z (通常物理模拟尝试在移动对象的同时旋转对象，但游戏中的角色不像正常的物体那样移动)和 Gravity Scale 0(稍后会重置，但现在不需要重力)。

玩家实体现在可以使用控制移动的脚本了。

6.2.1　编写键盘控制

首先，让玩家左右移动。垂直移动在平台游戏中也很重要，稍后再处理它。在 Scripts 文件夹中创建一个称为 PlatformerPlayer 的 C#脚本，然后拖放到场景中的 Player 对象上。

打开脚本，编写代码清单 6.1 中的代码。

代码清单 6.1　使用箭头键移动的 PlatformerPlayer 脚本

```
using UnityEngine;
using System.Collections;

public class PlatformerPlayer : MonoBehaviour {
  public float speed = 4.5f;

  private Rigidbody2D _body;

  void Start() {
    _body = GetComponent<Rigidbody2D>();      ← 需要把这个组件关联到 GameObject 上
  }

  void Update() {
    float deltaX = Input.GetAxis("Horizontal") * speed;
    Vector2 movement = new Vector2(deltaX, _body.velocity.y);      ← 仅设置水平移动，保留预先存在的垂直移动
    _body.velocity = movement;
  }
}
```

写完代码后，单击 Play，就可以用箭头键移动玩家了。该代码与前几章中的移动代码非常相似，主要区别在于它作用于 Rigidbody2D 而不是角色控制器。角色控制器是用于 3D 的，对于 2D 游戏，使用 Rigidbody 组件。注意，移动应用于 Rigidbody 的 velocity。

提示　默认情况下，Unity 会对箭头键输入施加一点加速度。不过，对于平台游戏来说，可能感觉不到这点加速度。对于更快速的控制，应把 Sensitivity 和 Gravity of Horizontal 输入增加到 6。要找到这些设置，请转到 Edit | Project Settings | Input，该列表很长，但是 Horizontal 是列表的第一个部分。

这是水平运动的主要方式！下面只需要处理碰撞检测。

6.2.2　与墙壁碰撞

注意，玩家现在能穿过街区。在地板或街区上没有碰撞器，所以玩家可以穿街而

过。为了解决这个问题，在地板和街区中添加 Box Collider 2D：选择场景中的每个对象，在 Inspector 中单击 Add Component，选择 Physics 2D | Box Collider 2D。

这就是需要执行的操作！现在单击 Play，玩家将无法穿越街区。就像第 2 章移动玩家一样，如果直接调整了玩家的位置，碰撞检测就无法工作，但是如果将移动应用到玩家的物理组件中，Unity 内置的碰撞检测就可以工作。换句话说，移动 Transform.position 将会忽略碰撞检测，因此应在 movement 脚本中操作 Rigidbody2D.velocity。

在更复杂的美术作品中加入碰撞器稍微复杂一些，但坦率地说，在这种情况下不会太难。即使美术作品不是一个精确的矩形，仍然可以使用 Box 碰撞器大致包围场景中障碍物的形状。另外，还有许多其他的碰撞器形状可以尝试，包括任意定制的多边形形状。图 6-6 说明了如何使用多边形碰撞器来处理形状奇怪的物体。

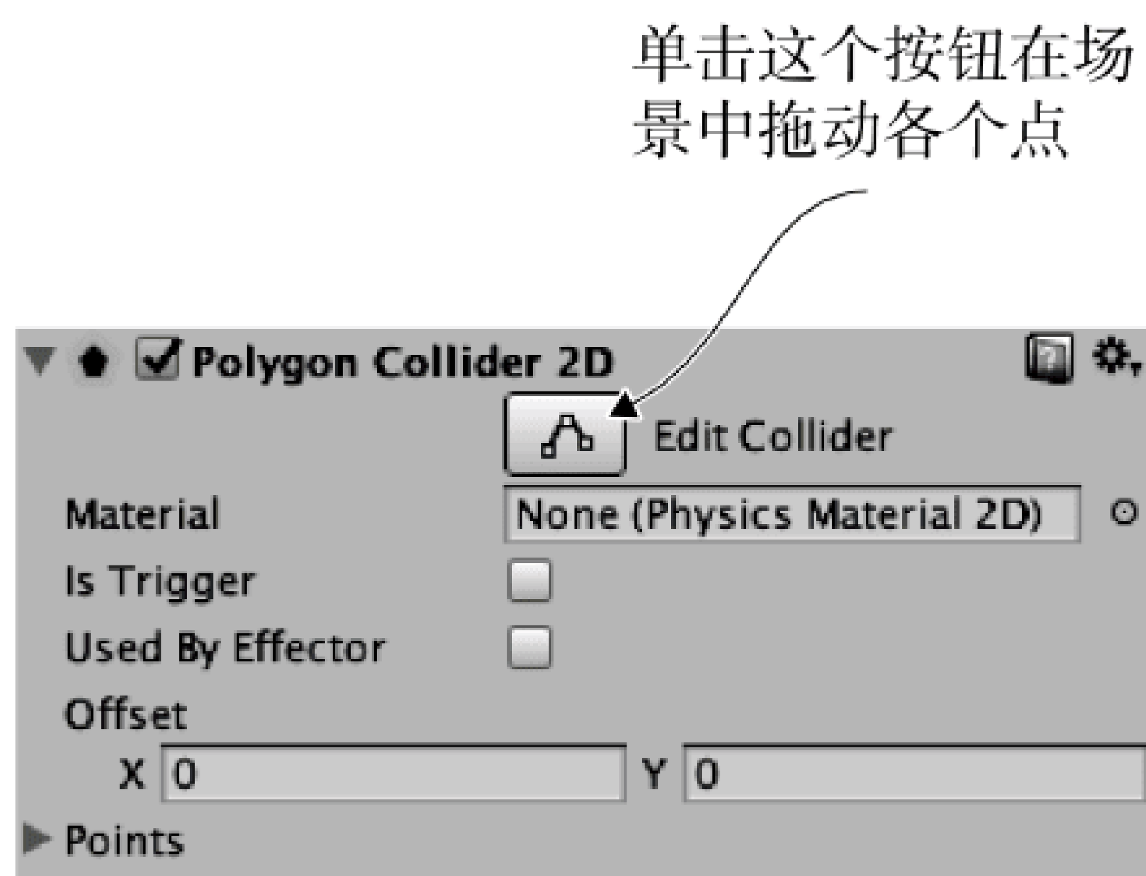


图 6-6 用 Edit Collider 按钮编辑多边形碰撞器的形状

无论如何，碰撞检测现在正在工作，所以下一步是让玩家随着它的移动而连续移动。

6.3 播放精灵动画

导入 stickman.png 时，将其分割成多个帧，以进行动画制作。现在播放这个动画，这样玩家就不会四处滑动，而是看起来像在行走。

6.3.1 讲解 Mecanim 动画系统

如第 4 章所述，Unity 中的动画系统称为 Mecanim。它的设计目的是可以直观地为角色建立复杂的动画网络，用最少的代码控制这些动画。这个系统对于 3D 角色非常有用(因此，在以后的章节中更详细地介绍它)，对于 2D 角色也是有用的。

动画系统的核心由两种不同的资源组成：动画剪辑和动画控制器(注意动画和动画控制器)。剪辑是要播放的单个动画循环，而控制器是控制何时播放动画的网络。这个网络是一个状态机图，图中的状态是可以播放的不同动画。控制器根据所观察的条件

在不同状态之间切换，并在每个状态下播放不同的动画。

将 2D 动画拖曳到场景中时，Unity 会自动创建这两种资源。也就是说，把动画的帧拖到场景中时，Unity 会自动创建使用这些帧的动画剪辑和动画控制器。如图 6-7 所示，展开精灵资源的所有帧，选择 frames 0-1，将它们拖曳到场景中，并在确认窗口中键入名称 stickman_idle。

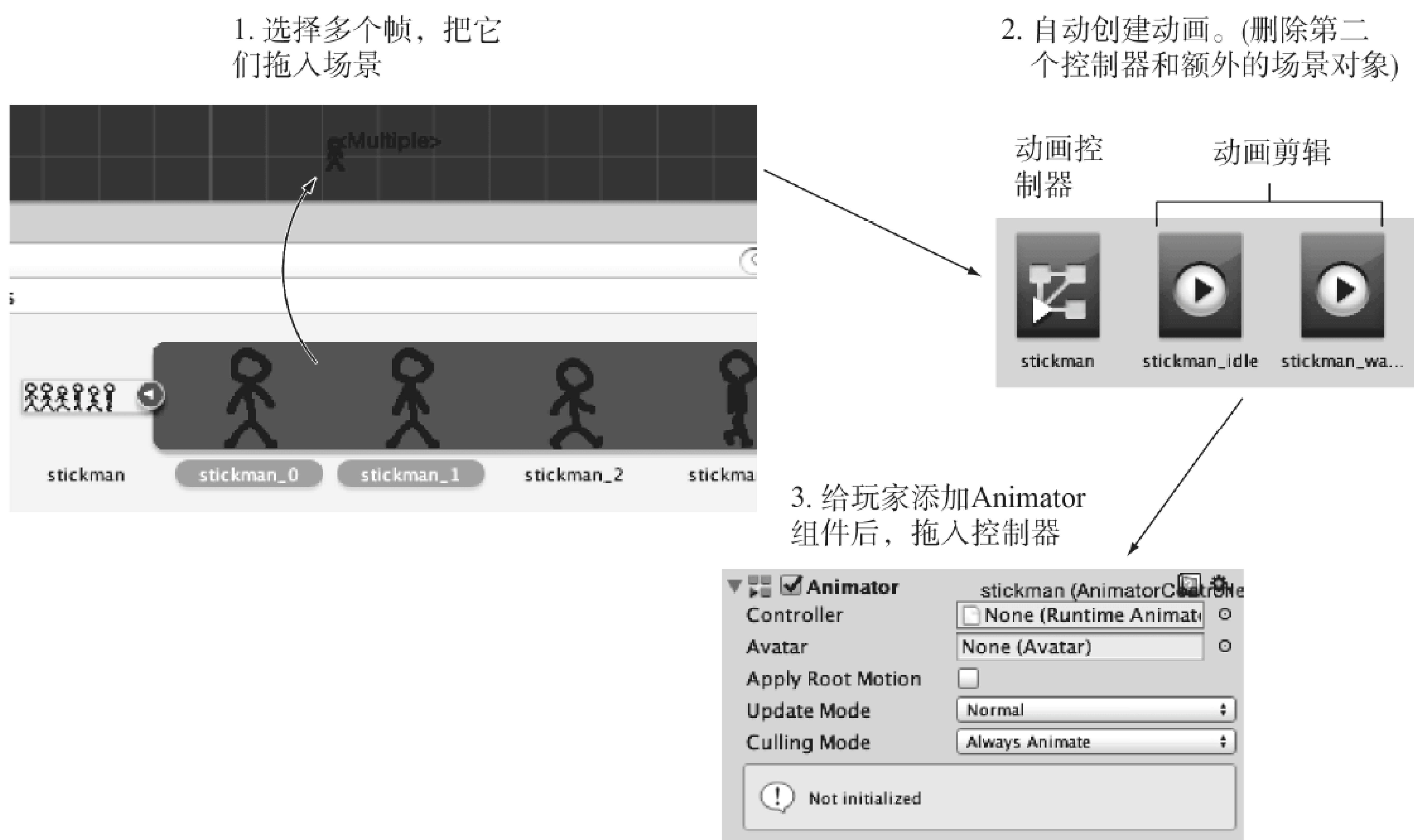


图 6-7　从精灵表帧进入玩家 Animator 的步骤

在 Asset 视图中，创建了一个剪辑 stickman_idle 和一个控制器 stickman_0，把控制器重命名为不带后缀的 stickman。很好，这样就创建了角色空闲时的动画!还在场景中创建了一个对象 stickman_0，但本例不需要它，因此删除它。

现在为步行动画重复这个过程。选择帧 2-5，将它们拖曳到场景中，并将动画命名为 stickman_walk。这次，删除场景中的 stickman_2 和资源中的控制器。只需要一个动画控制器控制两个动画剪辑，所以保留第一个控制器，删除新建的 stickman_2。

要将控制器应用于玩家角色，在场景中选择 Player 并添加 Miscellaneous > Animator 组件。如图 6-7 所示，将 stickman 控制器拖动到 Inspector 的控制器槽中。仍然选中玩家，打开 Window | Animator(如图 6-8 所示)。

Animator 窗口中的动画显示为块，称为状态，控制器在运行时在状态之间切换。这个控制器中已经有空闲状态，但是现在需要添加一个行走状态，将 stickman_walk 动画剪辑从 Assets 拖到 Animator 窗口中。

默认情况下，空闲动画的播放速度过快，所以将空闲动画的速度降低到 0.2。选择空闲动画状态，在右侧面板中会看到速度设置。有了这个更改，动画就都设置好了，可以进行下一步了。

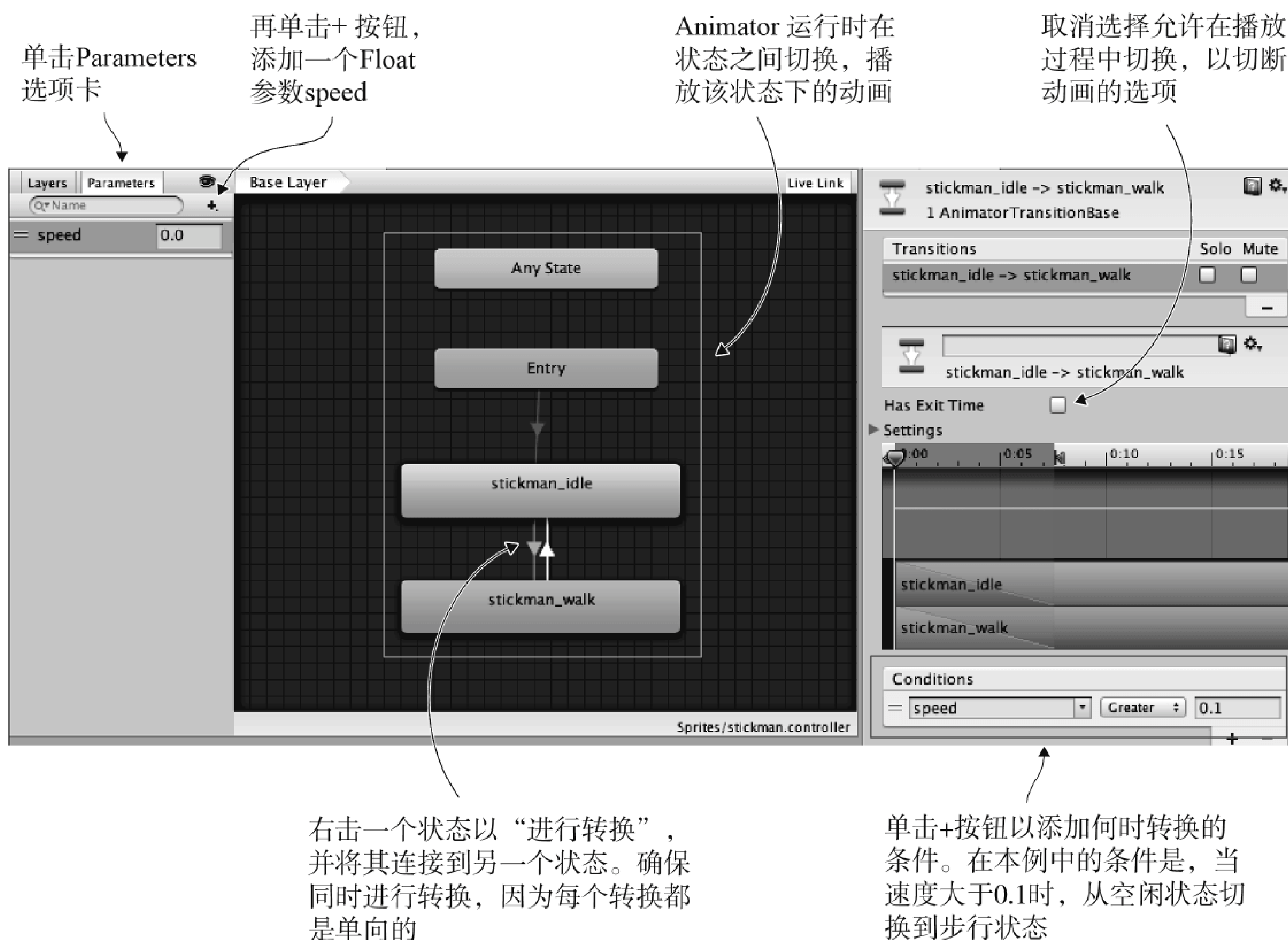


图 6-8　Animator 窗口，显示动画状态和转换

6.3.2　在代码中触发动画的播放

现在，已经在 Animator 控制器中设置了动画状态，可以在这些状态之间切换来播放不同的动画。如上一章所述，状态机根据所观察到的条件进行状态切换。在 Unity 的动画控制器中，这些条件称为参数，下面就添加一个参数。图 6-8 指出了相关的控件，选择 Parameters 选项卡，单击+按钮可以看到不同参数类型的菜单。添加一个名为 speed 的浮点参数。

接下来，需要基于该参数在动画状态之间切换。右击 idle，选择 Make Transition，这将开始从空闲状态中拖出一个箭头。单击 walk 连接到该状态，由于转换是单向的，因此右击 walk，切换回空闲状态。

现在选择从 idle 开始的转换(可以单击箭头本身)，取消选择 Has Exit Time，然后单击底部的+，以添加条件。建立条件 speed Greater than .2，这样状态就会在该条件下发生转变。现在对从 walk 到 idle 的转换再建立一次条件：选择从 walk 开始的转换，取消选中 Has Exit Time，添加一个条件，建立条件 speed Less than .2。

最后，movement 脚本可以操作动画控制器，如代码清单 6.2 所示。

代码清单 6.2　在移动过程中触发动画

```
...
private Animator _anim;
...
void Start() {
  _body = GetComponent<Rigidbody2D>();
  _anim = GetComponent<Animator>();
}

void Update() {
  ...
  _anim.SetFloat("speed", Mathf.Abs(deltaX));
  if (!Mathf.Approximately(deltaX, 0)) {
    transform.localScale = new Vector3(Mathf.Sign(deltaX), 1, 1);
  }
}
...
```

现有代码帮助显示新代码的位置

即使速度为负，Speed 也大于零

浮点数并不总是精确的，所以使用 Approximately()进行比较

移动时，按正 1 或负 1 的比例向左或向右移动

哇，这就是控制动画的代码！大部分工作都由 Mecanim 处理，操作它只需要少量代码。运行游戏，四处移动，观看玩家精灵的动画。这个游戏就要完成了，下面执行下一个步骤吧！

6.4　添加跳跃功能

玩家可以来回移动，但还不能垂直移动。垂直移动(既可以从平台的边缘上跳下来，也可以跳到更高的平台上)是平台游戏的一个重要部分，下面就来实现它。

6.4.1　因重力而下落

与直觉相反的是，在让玩家跳跃之前，它需要重力才能跳跃。之前在玩家的 Rigidbody 上把 Gravity Scale 设置为 0。这样玩家就不会因为重力而下落。现在把 Gravity Scale 设置回 1：选择场景中的 Player 对象，在 Inspector 中找到 Rigidbody，然后在 Gravity Scale 中输入 1。

重力现在会影响玩家，但是(假设在 Floor 对象上添加了一个 Box Collider)地板支撑着玩家。玩家走到地板的边缘就会掉落下来。默认情况下，重力对玩家的影响有点弱，所以需要增加它的影响。物理模拟有一个全局重力设置，可以在 Edit 菜单中调整它。具体来说，进入 Edit | Project Settings | Physics 2D。如图 6-9 所示，在各

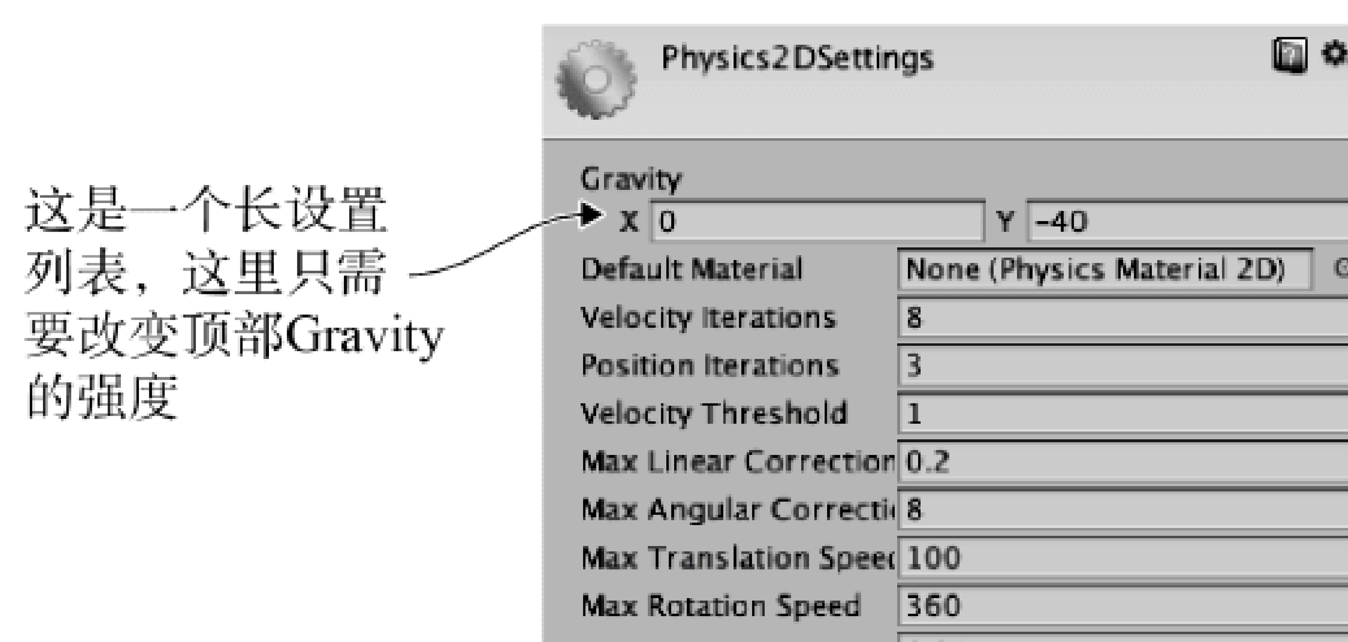


图 6-9　Physics 设置中的 Gravity 强度

种控件和设置的顶部，应该会看到 Gravity Y，把它更改为-40。

注意一个微妙的问题：下落的玩家会粘在地板边缘。要看到这个问题，可以让玩家走下平台边缘，立即从另一个方向回到平台。幸运的是，Unity 很容易修复这个问题。只需要给 Block 和 Floor 添加 Physics 2D > Platform Effector 2D 组件。这个效应器使场景中的对象表现得更像平台游戏中的平台。图 6-10 指出了需要调整的两个设置：设置 Collider 上的 Used By Effector，关闭 Effector 上的 Use One Way(后一种设置将在其他平台上使用，但现在不使用)。

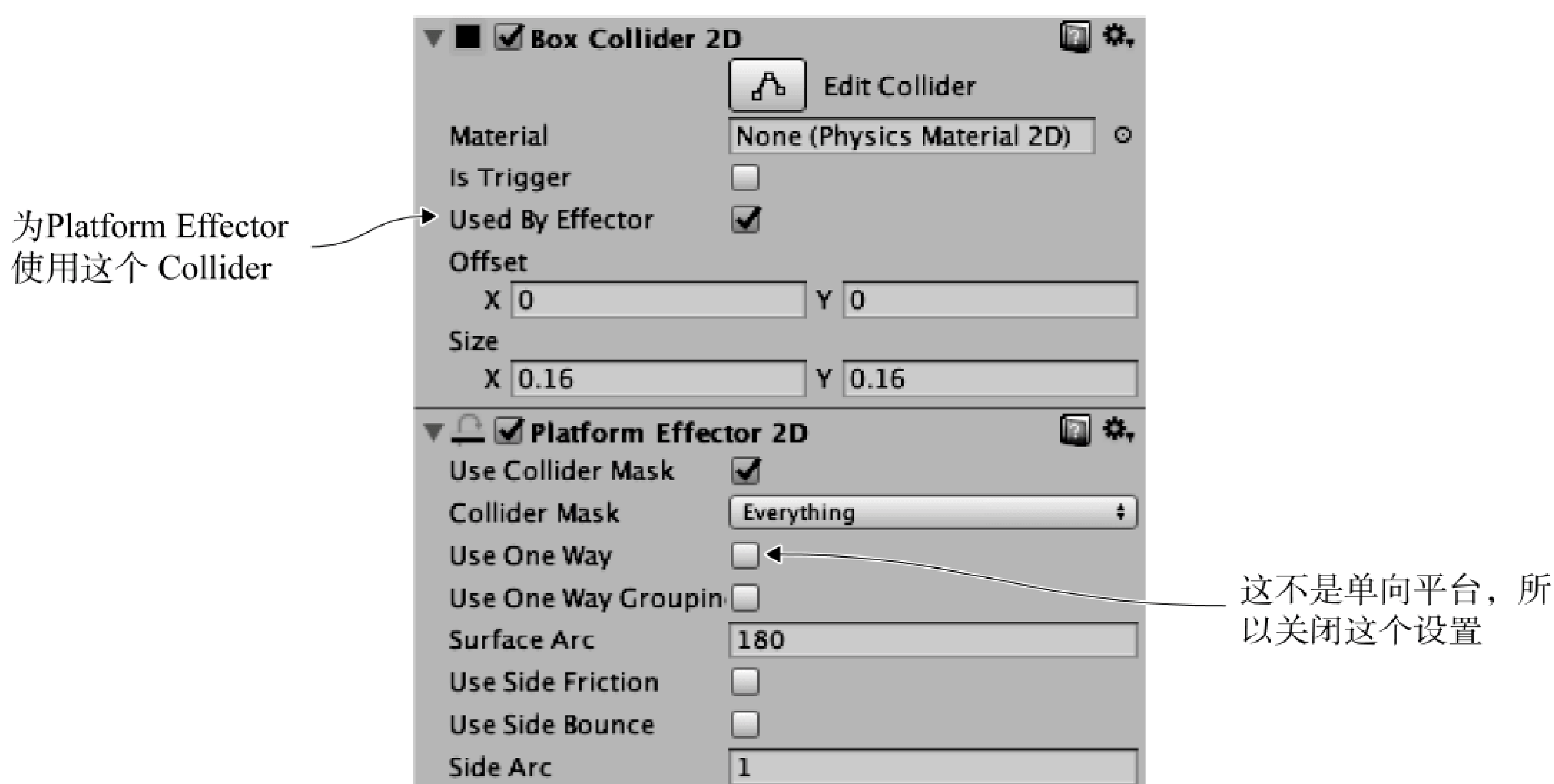


图 6-10　Inspector 中的 Collider 和 Effector 设置

这就解决了垂直运动的向下部分，但仍然需要解决向上部分。

6.4.2　施加向上的跃动

下一个需要添加的动作是跳跃。当玩家单击 Jump 按钮时，会产生向上的跃动。虽然代码直接改变了水平运动的速度，但是垂直速度保持不变，这样重力就可以起作用了。另外，物体会受到重力以外的其他力的影响，所以增加一个向上的力。将此代码添加到 movement 脚本中。(见代码清单 6.3)

代码清单 6.3　按空格键时跳跃

```
...
public float jumpForce = 12.0f;        // 现有代码帮助显示新代码的位置
...
_body.velocity = movement;

if (Input.GetKeyDown(KeyCode.Space)) {        // 仅在按空格键时添加力
  _body.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
}
...
```

重要的代码行是 AddForce()命令。该代码向 Rigidbody 增加向上的力，并在脉冲模式下这样做；脉冲是一种临时的作用力，而不是连续的作用力。这样，当按下空格键时，代码就会施加一个向上的临时作用力。同时，重力继续影响跳起的玩家，玩家跳跃的结果是得到一个漂亮的弧线。

但是，还有一个问题，下面来解决这个问题。

6.4.3　检测地面

跳跃控制有一个微妙的问题：玩家可以在半空中起跳！如果玩家已经在半空中(要么是因为玩家已经跳起来了，要么是因为玩家正在下落)，按下空格键会施加向上的力，但此时不应该施加向上的力。相反，跳跃控制应该只在玩家在地面上的时候工作。因此需要检测玩家是否在地面上。(见代码清单 6.4)

代码清单 6.4　检测玩家是否在地面上

```
...
private BoxCollider2D _box;
...
_box = GetComponent<BoxCollider2D>();        // 让这个组件使用玩家的碰撞器作为检查区域
...
_body.velocity = movement;

Vector3 max = _box.bounds.max;
Vector3 min = _box.bounds.min;
Vector2 corner1 = new Vector2(max.x, min.y - .1f);   // 检查碰撞器的最小 Y 值
Vector2 corner2 = new Vector2(min.x, min.y - .2f);

Collider2D hit = Physics2D.OverlapArea(corner1, corner2);

bool grounded = false;
if (hit != null) {        // 如果在玩家下方检测到碰撞器
  grounded = true;
}

if (grounded && Input.GetKeyDown(KeyCode.Space)) {     // 在跳跃条件中添加接地
...
```

有了这些代码，玩家就不能在半空中跳跃了。添加的这个脚本检查了玩家下方的碰撞器，并在跳跃的条件语句中考虑它。具体来说，代码首先获取玩家碰撞框的边界，然后在玩家下方相同宽度的区域内寻找重叠的碰撞器。该检查的结果存储在 grounded 变量中，并在条件中使用。

6.5　平台游戏的附加功能

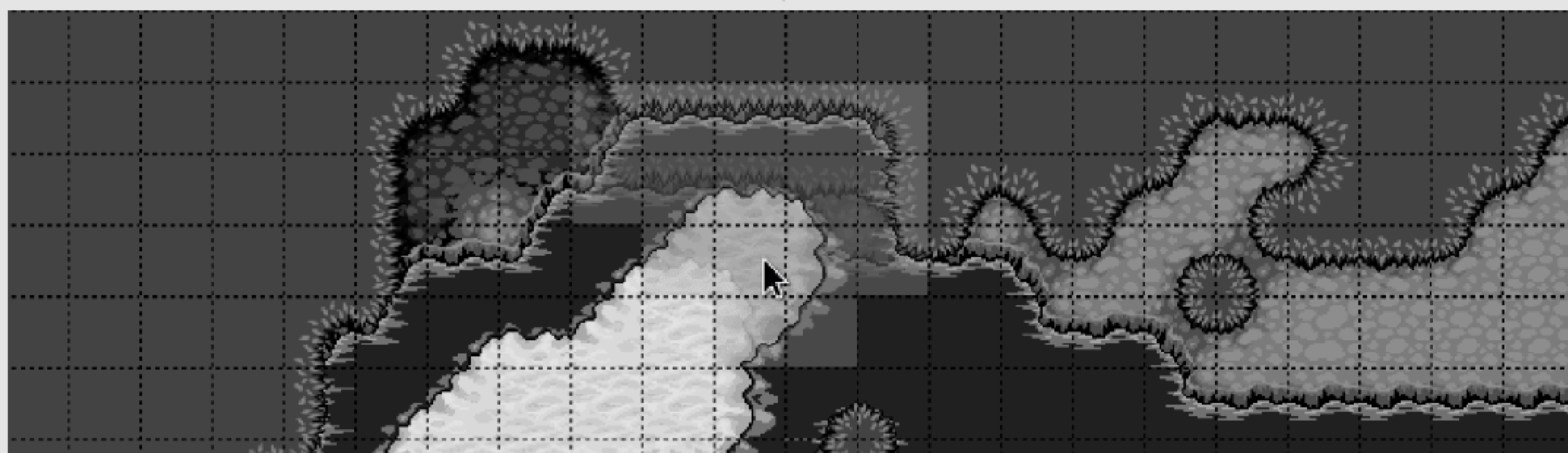
目前实现了玩家移动最关键的方面：步行和跳跃。下面围绕玩家的环境添加新功

能，来完成这个平台游戏的演示。

使用 tilemap 设计关卡

在项目中，地面和平台都是空白的白色矩形，而完成的游戏应该有更好的图形，但是关卡大小的图像对于计算机来说太大了，无法处理。这个问题最常见的解决方案是使用 tilemap。这种技术用大量平铺的小图像构建更大的组合图像。如图 6-11 所示的这张图片展示了一个 tilemap 的例子。

模糊的网格线显示贴图的边界，
这个网格不在实际的地图中



(image courtesy of mapeditor.org, using tiles from lpc.opengameart.org)

图 6-11　展示了一个 tilemap 的例子

注意地图是由小块图像构建的，这些小块图像在整个地图中都是重复出现的。这样，任何一幅图像都不大，但整个屏幕可以覆盖自定义的艺术品。Unity 最新版本包含其官方的 tilemap 系统。也可以使用外部库，比如 Tiled2Unity，这是一个 tilemap 系统，它导入了在 Tiled 中创建的 tilemap，Tiled 是一个非常流行的(免费)tilemap 编辑器。

包含更多信息的网站有 http://mng.bz / 318f 和 www.seanba.com/ tiled2unity。

6.5.1　不同寻常的楼层：斜坡和单向平台

现在，这个演示游戏的楼层是普通的、水平的，可以站立。不过，平台游戏常常使用许多有趣的平台，因此下面实现一些其他选项。要创建的第一个不同寻常的楼层是斜坡：复制 Floor 对象，将副本的旋转设置为(0, 0, −25)，将其移到左侧，大约位置是(−3.47, −1.27, 0)，命名为 Slope。

如果现在玩这个游戏，玩家在移动时就能正确地上下滑动了，但是空闲时，由于重力作用，会慢慢地下滑。为了解决这个问题，在玩家(a)站在地上，且(b)空闲时，为玩家关闭重力。幸好，前面已经检测了地面，因此可以在新代码中重用它。实际上，只需要一行新代码。(见代码清单 6.5)

代码清单 6.5　玩家站在地上时，关闭重力

```
...
_body.gravityScale = grounded && Mathf.Approximately(deltaX , 0, ? 0 : 1;
if (grounded && Input.GetKeyDown(KeyCode.Space)) {
...
```

检查玩家站在地上，且没有移动

现有代码帮助显示新代码的位置

通过对移动代码的调整，玩家角色已经可以正确地导航斜坡了。其次，单向平台是平台游戏中另一种不同寻常的地板。这是指可以在平台上跳跃，但仍然站在上面；玩家把头撞在完全坚固的普通平台底部。

因为单向平台在平台游戏中很常见，所以 Unity 为它们提供了功能。前面添加 Platform Effector 组件时，有一个单向设置被关闭。现在打开它！要创建一个新平台，复制 Floor 对象，该副本缩放为(10, 1, 1)，放在地板上方，位置为(−1.68, 0.11, 0)，并命名为Platform。别忘了打开Platform Effector组件的Use One Way选项。

玩家从下面跳过平台，但是当从上面下来的时候站在平台上面。可能有一个要修复的问题，如图 6-12 所示。Unity 可能会在玩家的上面显示平台精灵(为了更容易看到它，将 Jump Force 设置为 7 进行测试)，但我们希望玩家显示在最上面。可以调整玩家的 Z 坐标，但这次要调整其他内容来显示另一个选项。精灵渲染器有一个排序顺序，可以用来控制哪个精灵出现在最上面。在玩家的 Sprite Renderer 组件中，将 Order in Layer 设置为 1。

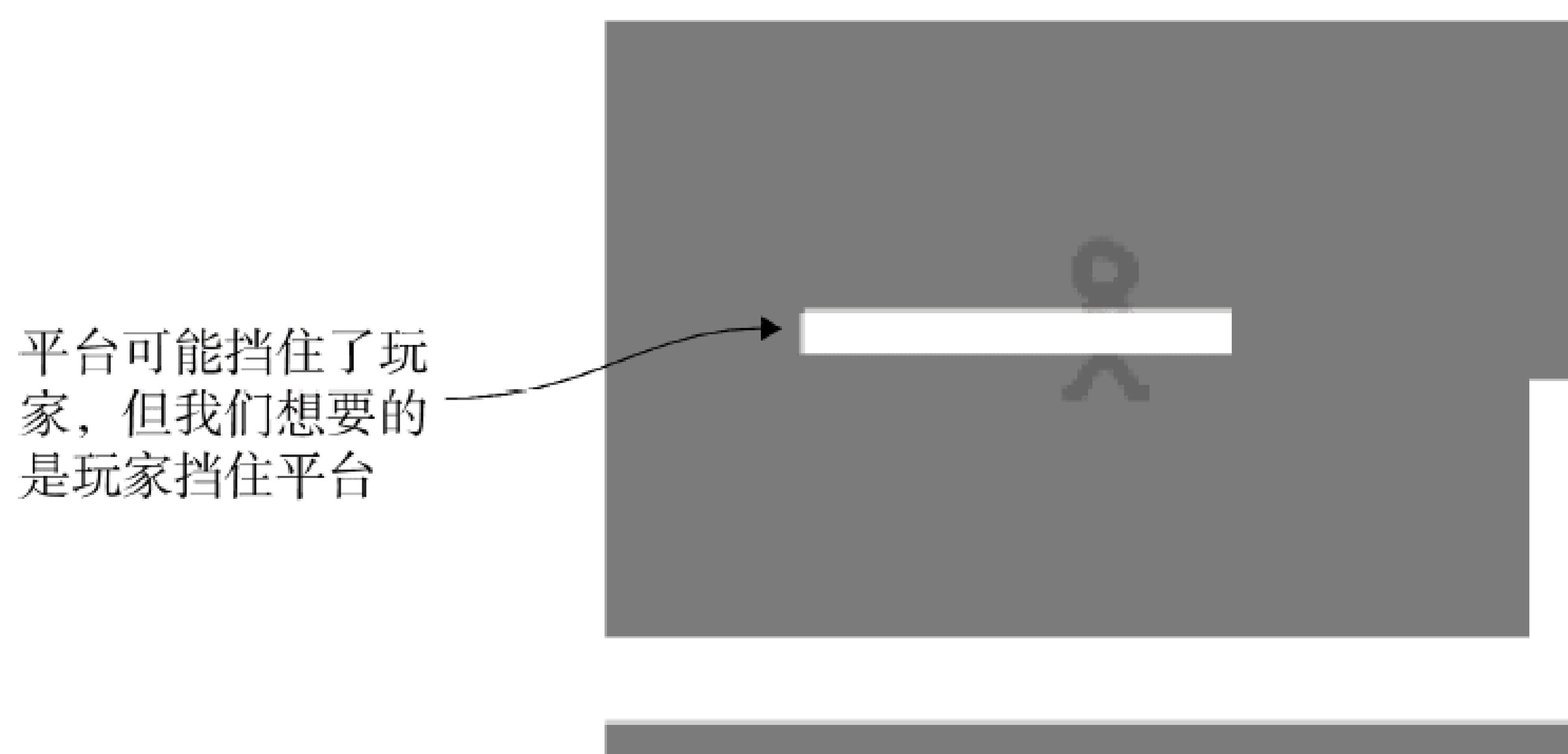


图 6-12　平台精灵挡住了玩家精灵

这包括倾斜的地板和单向平台。后面还会讲到另外一种不同寻常的楼层，但实现起来要复杂得多。

6.5.2　实现移动的平台

平台游戏中常见的第三种不同寻常的楼层是移动平台。实现它需要一个新的脚本来控制平台本身，也需要修改玩家的移动脚本来处理移动平台。下面编写一个脚本，

该脚本采用两个位置 start 和 finish，并使平台在它们之间来回移动。首先，创建一个新 C#脚本 MovingPlatform，并在其中编写代码。(见代码清单 6.6)

代码清单 6.6　来回移动的楼层的 MovingPlatform 脚本

```
using UnityEngine;
using System.Collections;

public class MovingPlatform : MonoBehaviour {
  public Vector3 finishPos = Vector3.zero;        ← 要移动到的位置
  public float speed = 0.5f;

  private Vector3 _startPos;
  private float _trackPercent = 0;                ← 在 start 和 finish 之间“跟踪”多远
  private int _direction = 1;                     ← 当前移动的方向

  void Start() {
    _startPos = transform.position;               ← 在场景中从这个地方开始移动
  }

  void Update() {
    _trackPercent += _direction * speed * Time.deltaTime;
    float x = (finishPos.x - _startPos.x) * _trackPercent + _startPos.x;
    float y = (finishPos.y - _startPos.y) * _trackPercent + _startPos.y;
    transform.position = new Vector3(x, y, _startPos.z);

    if ((_direction == 1 && _trackPercent > .9f) ||
    (_direction == -1 && _trackPercent < .1f)) {  ← 在开始和结束时改变方向
      _direction *= -1;
    }
  }
}
```

绘制自定义的 Gizmos

我们编写的大部分代码都是为了运行游戏，但是 Unity 脚本也会影响 Unity 的编辑器。Unity 中一个经常被忽视的特性是添加新菜单和窗口的功能。脚本也可以在 Scene 视图中绘制自定义辅助图像，这样的辅助图像称为 Gizmos。

我们已经熟悉一些 Gizmos，比如显示碰撞器的绿色盒子。这些都内置在 Unity 中，也可以在脚本中绘制自己的 Gizmos。例如，在 Scene 视图中画一条线，来显示平台的移动路径是很有用的，如图 6-13 所示。

画那条线的代码很简单。通常，当编写的代码影响 Unity 的编辑界面时，需要在顶部添加 using UnityEditor; (因为大多数编辑器函数都驻留在该名称空间中)，但是在本例中不需要它。将此方法添加到 MovingPlatform 脚本:

```
...
void OnDrawGizmos()
  { Gizmos.color = Color.red;
  Gizmos.DrawLine(transform.position, finishPos);
}
...
```

关于这段代码，有几点需要了解。首先，这些代码都在 OnDrawGizmos()方法中。与 Start 或 Update 一样，OnDrawGizmos 也是 Unity 识别的另一个方法名称。在该方法中有两行代码：一行设置绘图颜色，另一行告诉 Unity 从平台的当前位置到完成位置画一条线。

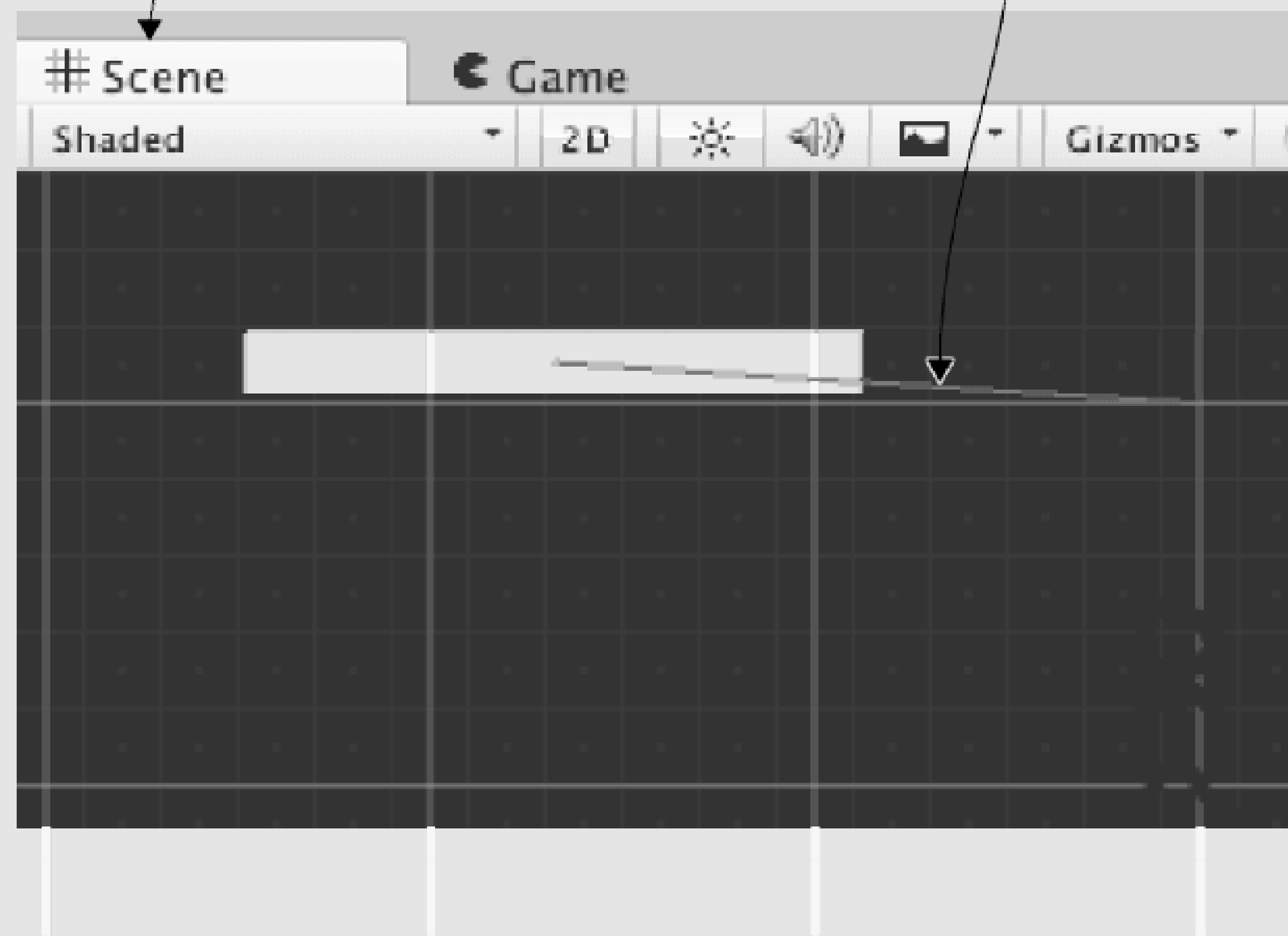


图 6-13　显示平台的移动路径是有用的

将此脚本拖放到平台对象上。现在运行游戏，平台就会左右移动！接着需要调整玩家的移动脚本，以便将玩家附加到移动平台。下面是要做的更改。

现在，玩家跳上平台后就会跟着平台一起移动。玩家大都关联为平台的子对象；记住，设置父对象时，子对象与父对象一起移动。代码清单 6.7 使用 GetComponent() 检查所检测的地面是否是一个移动平台。如果是，就将平台设置为玩家的父平台；否则，玩家将与任何父对象分离。

代码清单 6.7　在 PlatformerPlayer.cs 中处理移动的平台

```
...
   _body.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
}

MovingPlatform platform = null;
if (hit != null) {
   platform = hit.GetComponent<MovingPlatform>();     ← 检查玩家下方的平台是否是移动的平台
}
if (platform != null) {                               ← 连接平台或清除 transform.parent
   transform.parent = platform.transform;
} else {
   transform.parent = null;
}

_anim.SetFloat("speed", Mathf.Abs(deltaX));           ← 现有代码帮助显示新代码的位置
...
```

但有一个大问题：玩家继承了平台的缩放比例，导致玩家的大小不符合要求。这可以通过反缩放(将玩家缩小以对抗平台的放大)来解决。(见代码清单 6.8)

代码清单 6.8 校正玩家的比例

```
...
  _anim.SetFloat("speed", Mathf.Abs(deltaX));

  Vector3 pScale = Vector3.one;          ← 如果玩家不在移动的平台上，玩家的默认比例就是 1
  if (platform != null) {
    pScale = platform.transform.localScale;
  }
  if (! Mathf.Approximately(deltaX, 0)) {
    transform.localScale = new Vector3(
    Mathf.Sign(deltaX) / pScale.x, 1/pScale.y, 1);   ← 用新代码替换已有的比例
  }
}
...
```

反缩放的数学原理很简单：将玩家设置为 1 除以平台的缩放比例，然后让玩家的缩放比例乘以平台的缩放比例，得到的缩放比例是 1。这段代码中唯一棘手的部分是乘以移动值的符号。如前所述，玩家是根据移动方向翻转的。

现在完全实现了移动的平台。这个平台游戏的演示只需要最后一个步骤了。

6.5.3 摄像机控制

移动摄像机是要添加到这个 2D 平台游戏的最后一个功能。创建一个名为 FollowCam 的脚本，将其拖到摄像机中，然后在其中写入以下内容。(见代码清单 6.9)

代码清单 6.9 与玩家一起移动的 FollowCam 脚本

```
using UnityEngine;
using System.Collections;

public class FollowCam : MonoBehaviour {
   public Transform target;

   void LateUpdate() {
     transform.position = new Vector3(
     target.position.x, target.position.y, transform.position.z);   ← 改变 X 和 Y 时，Z 坐标保持不变
   }
}
```

编写完代码后，将 Player 对象拖到 Inspector中脚本的target槽。播放场景时，摄像机会四处移动，让玩家保持在屏幕中央位置。代码将目标对象的位置应用到摄像机上，并将玩家设置为目标对象。注意，方法名是 LateUpdate 而不是 Update，这是 Unity 识别的另一个名字。LateUpdate 也会执行每一帧，但是在更新每一帧之后执行。

摄像机在任何时候都能精确地与玩家一起移动，这有点不和谐。在大多数平台游戏中，摄像机都有着各种微妙而复杂的行为，当玩家四处移动时，突出关卡的不同部分。事实上，对于平面游戏来说，摄像机控制是一个非常深入的话题，尝试搜索“平台游戏，摄像机”，并查看所有结果。不过，在本例中，只需要让摄像机的移动更流畅，更和谐。代码清单 6.10 进行了这样的调整。

代码清单 6.10　使摄像机的移动更流畅

```
...
public float smoothTime = 0.2f;

private Vector3 _velocity = Vector3.zero;
...
void LateUpdate() {
  Vector3 targetPosition = new Vector3(
  target.position.x, target.position.y, transform.position.z);

  transform.position = Vector3.SmoothDamp(transform.position,
   targetPosition, ref _velocity, smoothTime);
}
...
```

改变 X 和 Y 时，Z 坐标保持不变

从当前位置流畅地转换到目标位置

主要的变化是调用了 SmoothDamp 函数，其他的更改(如添加 time 和 velocity 变量)都是为了支持该函数。这是 Unity 提供的一个函数，它可以让值平稳地转换为新的值。在本例中，值是摄像机和目标的位置。

摄像机现在和玩家一起平稳地移动。我们实现了玩家的移动以及几种不同的平台，现在也实现了摄像机控制。本章的项目已经完成了！

6.6　小结

- 精灵表是一种处理 2D 动画的常见方式。
- 游戏中的角色不像现实世界中的对象，必须相应地调整其物理规则。
- Rigidbody 对象可以通过应用作用力来控制，或通过设置其速度来直接控制。
- 2D 游戏中的关卡通常由 tilemap 构建。
- 简单的脚本可以使摄像机流畅地跟随着玩家。

第 7 章

在游戏中放置 GUI

本章涵盖：

- 比较旧 GUI 系统(Unity4.6 之前)和新 GUI 系统
- 创建用于界面的画布
- 通过锚点定位 UI 元素
- 为 UI 添加交互(按钮、滑动条等)
- 从 UI 广播和侦听事件

本章将为 3D 游戏构建 2D 界面。前面在建立第一人称射击演示游戏时，主要关注虚拟场景本身，但每个游戏除了进行游戏玩法的虚拟场景外，还需要一些抽象交互，显示一些信息。所有游戏都是这样，不管是 2D 游戏还是 3D 游戏，第一人称射击游戏是益智游戏。因此，虽然本章中的技术将用于 3D 游戏，但它们也适用于 2D 游戏。

这些抽象的交互显示称为 UI，更专业的术语是 GUI。GUI 指的是界面的可视化部分，例如文本和按钮(如图 7-1 所示)。从技术上说，UI 包括非图形控件，例如键盘或手柄，但人们说的“UI”经常指的就是图形部分。

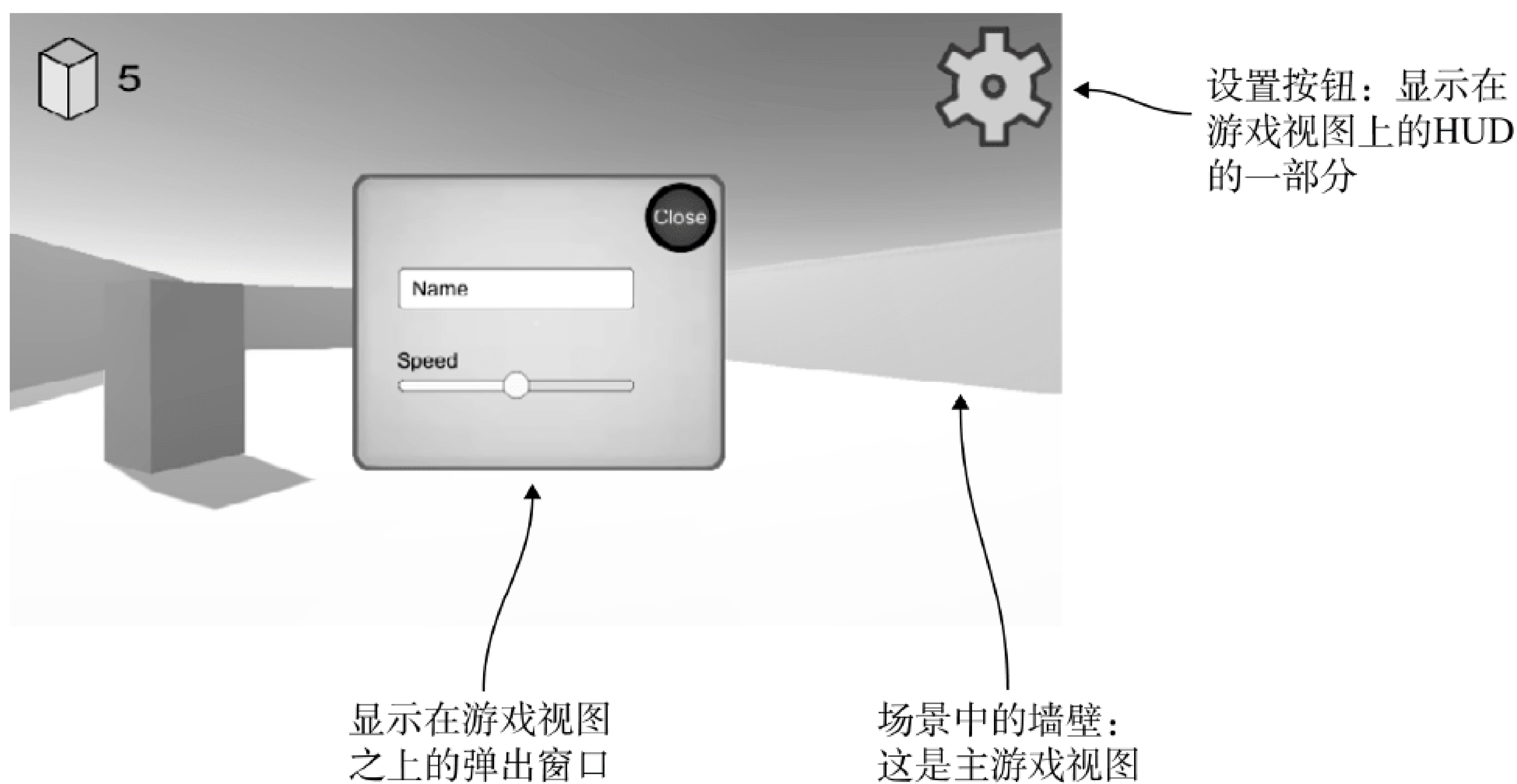


图 7-1　将为游戏创建的 GUI(平视显示，或 HUD)

尽管任何软件都需要一些 UI 才能让用户控制软件，但游戏通常以与其他软件稍微不同的方式使用自己的 GUI。以网站为例，GUI 基本就是网站(就视觉表现而言)。然而在游戏中，文本和按钮通常覆盖在游戏视图上，这是一种称为 HUD 的显示。

定义　平视显示(HUD，heads-up display)使图形叠加在世界视图上。这个概念源于军用飞机，目的是为了让飞行员不低头就能看到重要信息。类似的，叠加在游戏视图上的 GUI 称为 HUD。

本章将说明如何通过 Unity 最新的 UI 工具构建游戏的 HUD。如第 5 章所述，Unity 提供了多种创建 UI 显示的方式。本章演示了 Unity4.6 及其后续版本中新的 UI 系统。接下来讨论之前的 UI 系统和新系统的优势。

为了学习 Unity 中的 UI 工具，在第 3 章的第一人称射击(FPS)项目的基础上进行构建。本章中的项目包含以下步骤：

(1) 规划界面
(2) 放在显示界面中的 UI 元素
(3) 编写与 UI 元素的交互
(4) 让 GUI 响应场景中的事件
(5) 使场景响应 GUI 上的动作

注意　本章内容大部分情况下和基于构建的项目是独立的——它只是在已有的游戏演示上添加一个图形界面。本章的所有示例都构建于第 3 章的 FPS 之上，可以下载那个示例项目，也可以使用自己喜欢的游戏演示。

复制第 3 章的项目，打开副本，开始本章的工作。和往常一样，需要的美术资源可以在这个示例中下载。在准备好这些文件后，就可以开始构建游戏的 UI 了。

7.1　在开始写代码之前

在开始构建 HUD 之前，首先需要了解 UI 系统的工作原理。Unity 提供了多种方式来构建游戏的 HUD，因此接下来需要了解这些系统的工作原理。接着可以简单规划 UI，准备所需的美术资源。

7.1.1　立即模式 GUI 还是高级 2D 界面

Unity 从第一个版本开始就有了立即模式(immediate mode)GUI 系统。

定义　立即模式在每帧显式发出绘制命令。而对于另一种系统，只需要一次定义所有的视觉效果，之后系统就知道每帧需要绘制什么，而不必再重新声明。后一种方法称为保留模式(retained mode)。

立即模式系统可以简单地在屏幕上放置可单击的按钮。代码清单 7.1 展示了使用立即模式 GUI 系统的代码，只需要将这个脚本附加到场景中的任何对象上。对于另一个使用立即模式 GUI 的例子，可以回想第 3 章中显示的目标光标。这个 GUI 系统完全基于代码，不能在 Unity 的编辑器上工作。

代码清单 7.1　使用立即模式 GUI 创建按钮的示例

```
using UnityEngine;
using System.Collections;

public class BasicUI : MonoBehaviour {
    void OnGUI() {
       if (GUI.Button(new Rect(10, 10, 40, 20), "Test")) {
            Debug.Log("Test button");
       }
    }
}
```

函数在每帧渲染其他所有对象之后调用

参数：位置 X、Y、宽度、高度和文本标签

代码清单 7.1 中的核心代码是 OnGUI()方法。非常类似于 Start()和 Update()，每个 MonoBehaviour 自动响应 OnGUI()方法。这个方法会在每帧渲染完 3D 场景后执行，它提供了一个放置 GUI 绘制命令的地方。这段代码绘制了一个按钮；注意，用于按钮的命令会在每帧执行(这就是立即模式)。按钮命令在条件中使用，按钮被按下时可以进行响应。

由于立即模式 GUI 只需要做很少的工作就很容易将一些按钮添加到屏幕上，因此在后面章节的示例中使用它。但只有默认按钮才使用该系统创建，因此最新的 Unity 版本在编辑器中有一套基于 2D 图形的新界面系统。它需要更多的处理，但在成品游戏中可能使用更新的界面系统，因为它提供了更专业的效果。

新的 UI 系统以保留模式工作，因此图形只需要布局一次，就能在每帧绘制，而不需要重新定义。在这个系统中，用于 UI 的图形放在 Unity 的编辑器中。相比于立即

模式 UI，这提供了两个优势：①可以在放置 UI 元素时看到当前 UI 的外观。②这个系统很容易使用自己的图像来定制 UI。

为了使用这个系统，需要导入图像，并将对象拖动到场景中。接下来规划 UI 的外观。

7.1.2　规划布局

大多数游戏的 HUD 只是不停重复一些不同的 UI 控件。这意味着这个项目不需要学习构建非常复杂的 UI。接下来将在主游戏视图的屏幕角落中放置显示分数和设置按钮(如图 7-2 所示)。设置按钮将打开一个弹出窗口，该窗口中包含一个文本域和一个滑动条。

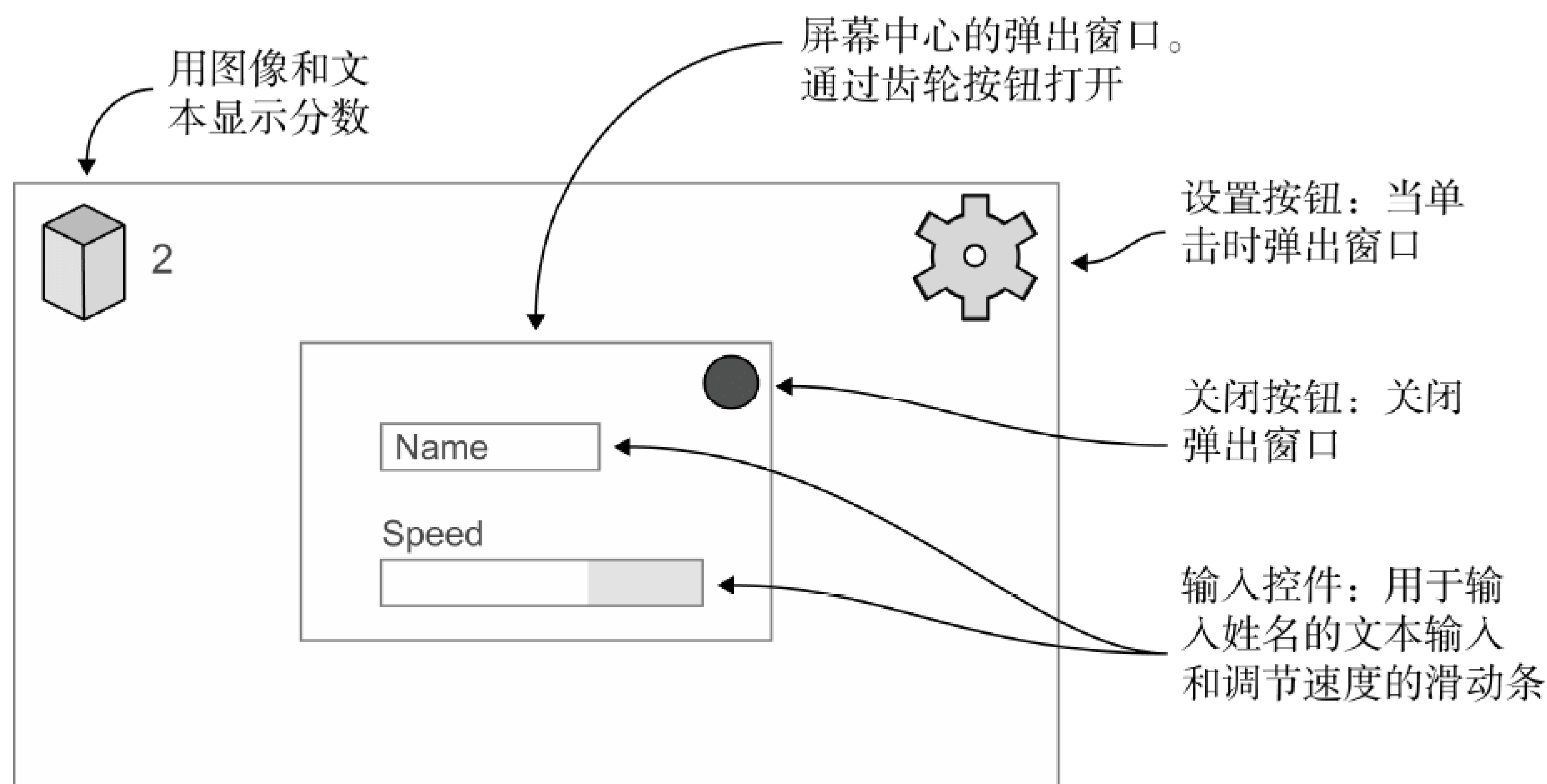


图 7-2　规划的 GUI

本例中，那些输入控件将用于设置玩家的姓名和移动速度，但基本上这些 UI 元素就可以控制游戏中的任何设置。

很好，规划确实比较简单！接下来将需要的图像导入到项目中。

7.1.3　导入 UI 图像

UI 需要一些图像来显示按钮之类的对象。下面使用类似第 5 章的 2D 图像构建 UI，因此需要遵循以下两个步骤：

(1) 导入图像(如果有必要，将它们设置为 Sprite)

(2) 将精灵拖动到场景中

为了完成这些步骤，首先将图像拖动到 Project 视图中，导入它们，接着在 Inspector 中将它们的 Texture Type 设置改为 Sprite(2D And UI)。

警告　Texture Type 设置在 3D 项目中默认为 Texture，而在 2D 项目中默认为 Sprite。如果要在 3D 项目中使用精灵，需要手动调整这个设置。

从下载的示例中获取需要的图像(如图 7-3 所示)，接着将图像导入到项目中。确保

所有导入的资源都被设置为 Sprite，可能需要在导入之后调整设置中的 Texture Type。



图 7-3　本章项目需要的图像

这些精灵构成了接下来将创建的按钮、分数显示和弹出窗口。现在图像已导入，下面将这些图形放到屏幕上。

7.2　设置 GUI 显示

美术资源和第 5 章使用的 2D 精灵是同一种资源，但在场景中，这些资源的用法稍有不同。Unity 提供了一些特殊的工具，使图像成为 HUD 并显示在 3D 场景上，而不是把图像显示为场景的一部分。UI 元素的定位通常有一些特殊技巧，因为显示可能需要根据不同的屏幕而变化。

7.2.1　为界面创建画布

UI 系统工作原理中最基础且最特殊的一面是所有图像必须附加到画布对象上。

提示　画布(Canvas)是一类特殊的对象，Unity 把它渲染为游戏的 UI。

打开 GameObject 菜单，查看可以创建的对象。在 UI 目录下，选择 Canvas。在场景中将会出现一个 canvas 对象(将该对象命名为 HUD Canvas 可能会更清晰)。该对象代表整个屏幕的范围，相对于 3D 场景，它非常大，因为它将屏幕上的一个像素缩放为场景中的一个单位。

警告　当创建 canvas 对象时，也会自动创建 EventSystem 对象。对于 UI 交互，该对象是必需的，但你可以忽略它。

切换为 2D 视图模式(见图 7-4)，双击 Hierarchy 中的画布，缩放它，使画布完全显示出来。当整个项目在 2D 中时，2D 视图模式会自动启用，但在 3D 项目中，需要手动单击才能在 UI 和主场景之间切换。为了切换回 3D 场景，关闭 2D 视图模式，并双击场景，使视野缩放到该对象。

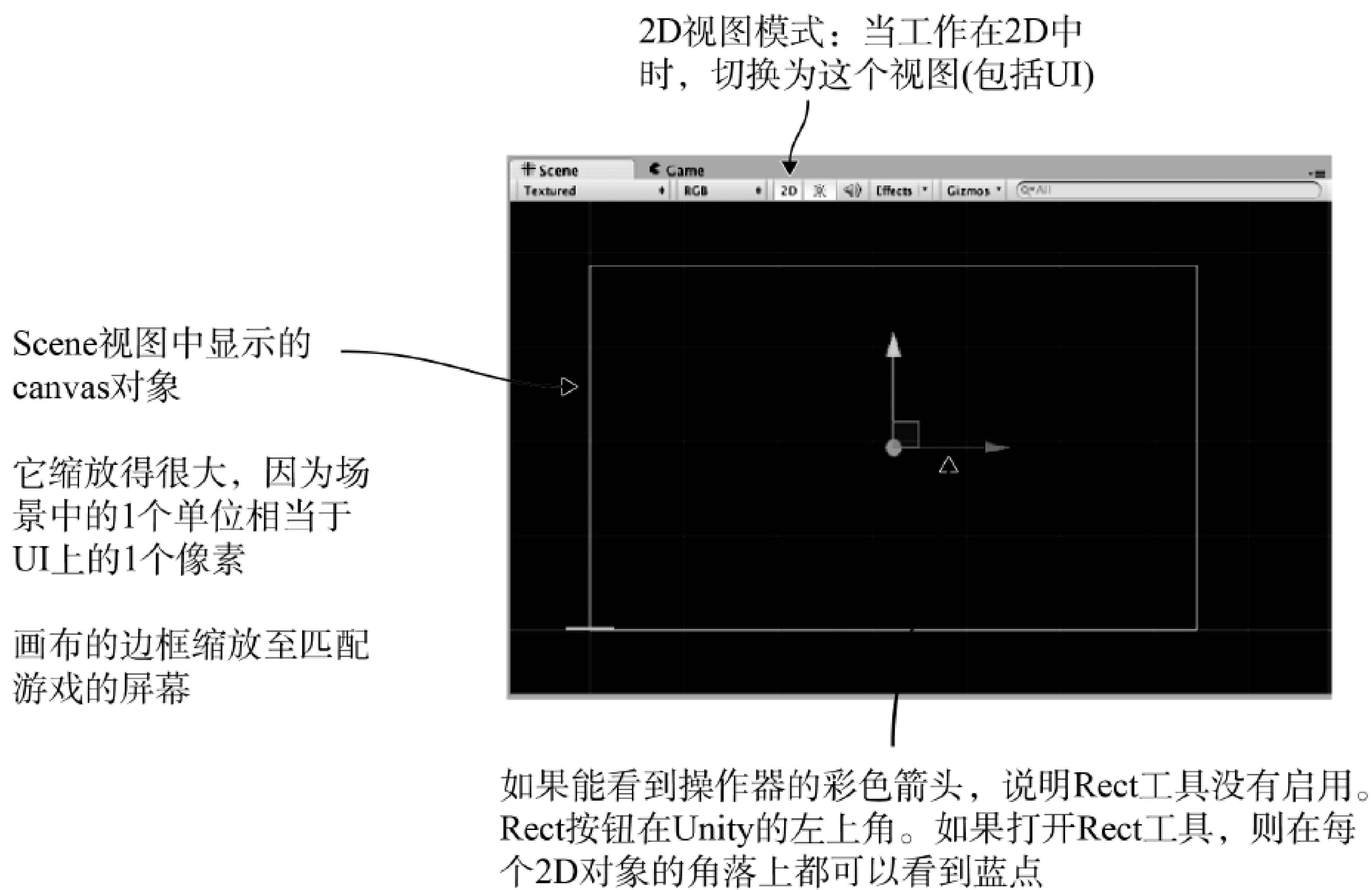


图 7-4　Scene 视图中的空画布对象

提示　不要忘记第 4 章的提示：Scene 视图面板顶部的按钮控制所显示的内容，因此查找 Effects 按钮，关闭天空盒。

画布中有一些可以调整的设置。首先是 Render Mode 选项，让它保持默认设置 Screen Space—Overlay，但应该知道如下三种设置的含义：

- Screen Space—Overlay：将 UI 渲染为摄像机视图顶部的 2D 图形(这是默认设置)。
- Screen Space—Camera：将 UI 渲染在摄像机视图顶部，但 UI 元素可以旋转，得到透视效果。
- World Space：将画布对象放在场景中，就好像 UI 是 3D 场景的一部分。

初始默认设置之外的另外两种模式有时对于实现特殊效果很有用，但也会稍微复杂一些。

另外一个重要的设置是 Pixel Perfect。这个设置会轻微调整图像的位置，以使图像非常清晰(相反，在像素之间定位时，图像会比较模糊)。选中该复选框。现在 HUD 画布已经设置完毕，但它依然是空白的，此时需要一些精灵。

7.2.2　按钮、图像和文本标签

画布对象定义了一个用于显示 UI 的区域，但它依然需要精灵来显示。如果采用图 7-2 中的 UI 版面，就会有个方块/敌人的图像在左上角，后面跟着显示分数的文本，右上角还有一个齿轮形状的按钮。因此，在 GameObject 菜单的 UI 部分，只要为每个元素创建图像、文本或按钮即可。

提示　类似第 5 章使用的文本对象，应该考虑在自己的项目中使用 Text-Mesh Pro。那个系统是外部开发的，以改进 Unity 的文本。

为了正确显示 UI 元素，UI 元素需要成为画布对象的子对象。Unity 自动处理了这个操作，但记住，通常可以在 Hierarchy 视图(如图 7-5 所示)中拖动对象来创建父子关系。

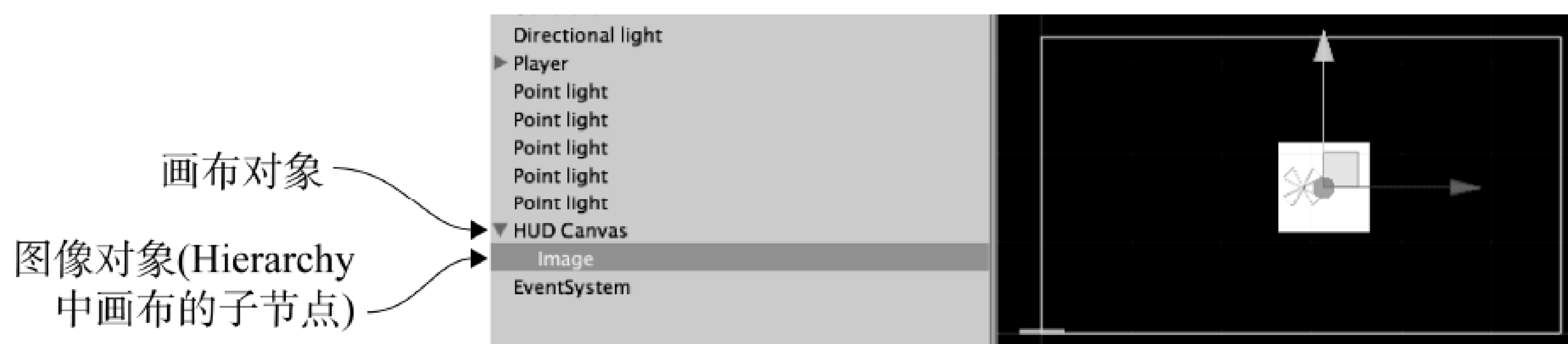


图 7-5　Hierarchy 视图中关联了图像的画布

画布中的对象可以因为定位而成为其他对象的父节点，就像场景中的任何对象一样。例如，把文本对象拖放到图像对象上，使文本随着图像一起移动。类似的，默认按钮对象把文本对象作为其子节点，这个按钮不需要文本标签，因此可以删除文本对象。

将 UI 元素粗略定位到角落，下一节将更精确地定位它们。现在拖动对象，直到将它们放在大致的位置。单击并将图像对象拖放到画布的左上角，将按钮移到右上角。

提示　如第 5 章所述，在 2D 模式下使用 Rect 工具。它是一个合并了三种变换(Move、Rotate 和 Scale)的工具。这些操作在 3D 模式中拆分为单独的工具，但在 2D 模式中合并起来，这是因为在 2D 模式中有一个维度不需要关心。在 2D 模式中，这个工具被自动选中，也可以单击 Unity 左上角的按钮来选中它。

此时，图像是空白的。如果选择一个 UI 对象，观察它的 Inspector，将会发现图像组件的顶部附近有一个 Source Image 槽。如图 7-6 所示，从 Project 视图中拖动精灵(记住不是贴图！)，将图像赋给对象。将敌人精灵赋给图像对象，将齿轮精灵赋给按钮对象(在将精灵赋到对象后，单击 Set Native Size 来修正图像对象的大小)。

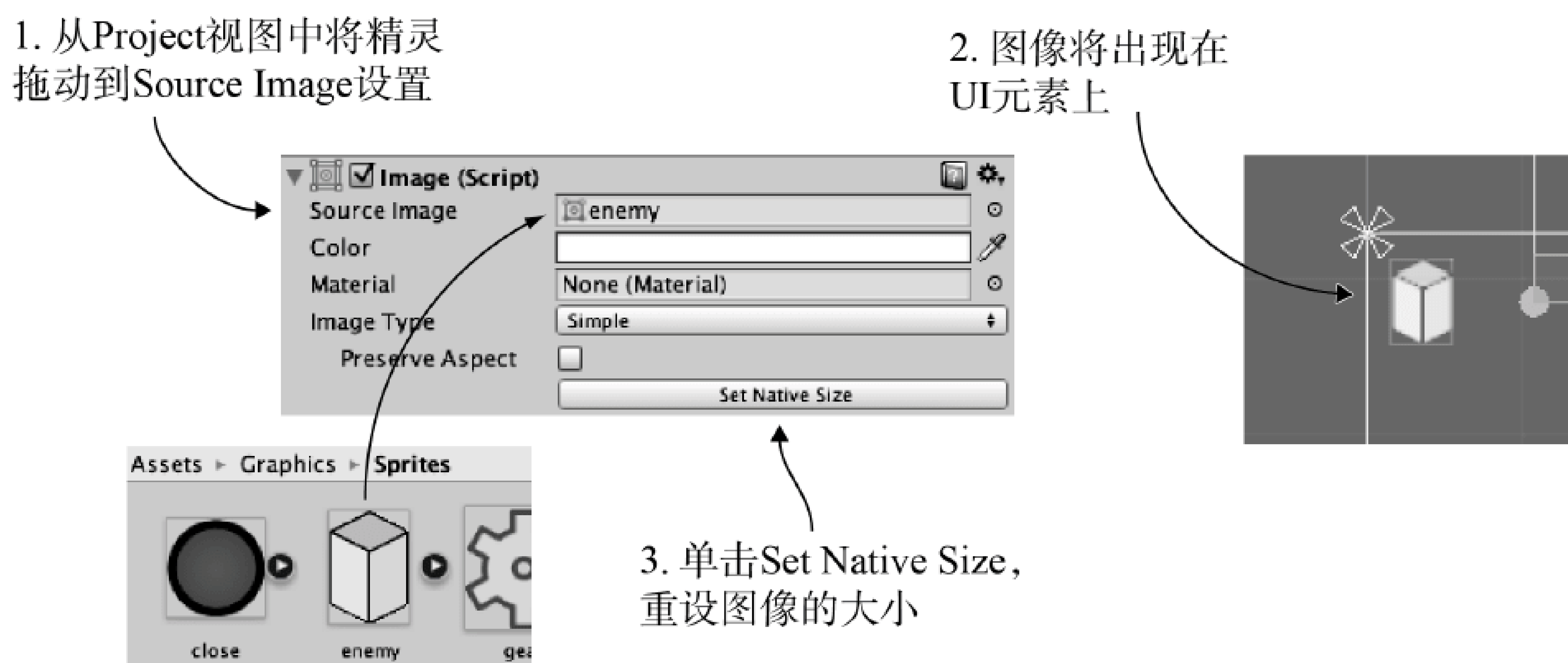


图 7-6　将 2D 精灵赋给 UI 元素的 Image 属性

首先关注敌人图像和齿轮按钮的外观。对于文本对象，在 Inspector 中有一些设置。首先，在大的 Text 框中输入一个数字，这个文本以后将被覆盖，但现在还是有用的，因为它看起来像是在编辑器中显示分数。由于该文本太小，因此增加 Font Size 为 24 并设置样式为 Bold。还可以将这个标签设置为水平左对齐(如图 7-7 所示)和垂直居中对齐。现在，剩余的设置保留它们的默认值即可。

提示 除了 Text 盒和对齐，最常调整的属性就是字体。可以将 TrueType 字体导入到 Unity 中，然后将该字体放到 Inspector 中。

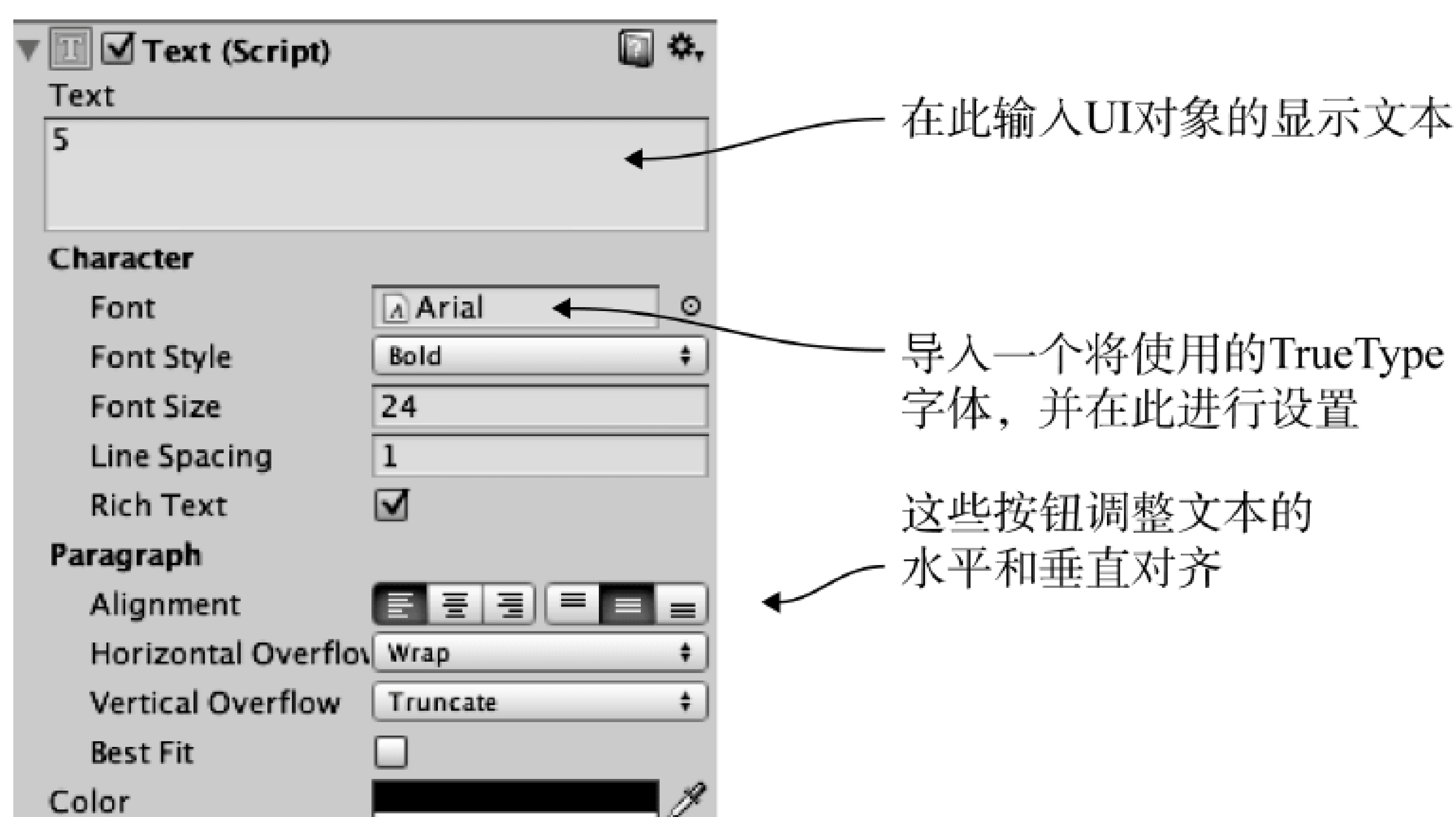


图 7-7　UI 文本对象的设置

现在精灵已经赋给 UI 图像，分数文本也已设置好，可以单击 Play 看看 3D 游戏顶部的 HUD。如图 7-8 所示，显示在 Unity 编辑器中的画布显示了屏幕的边界，可以在屏幕的这些位置上绘制 UI 元素。

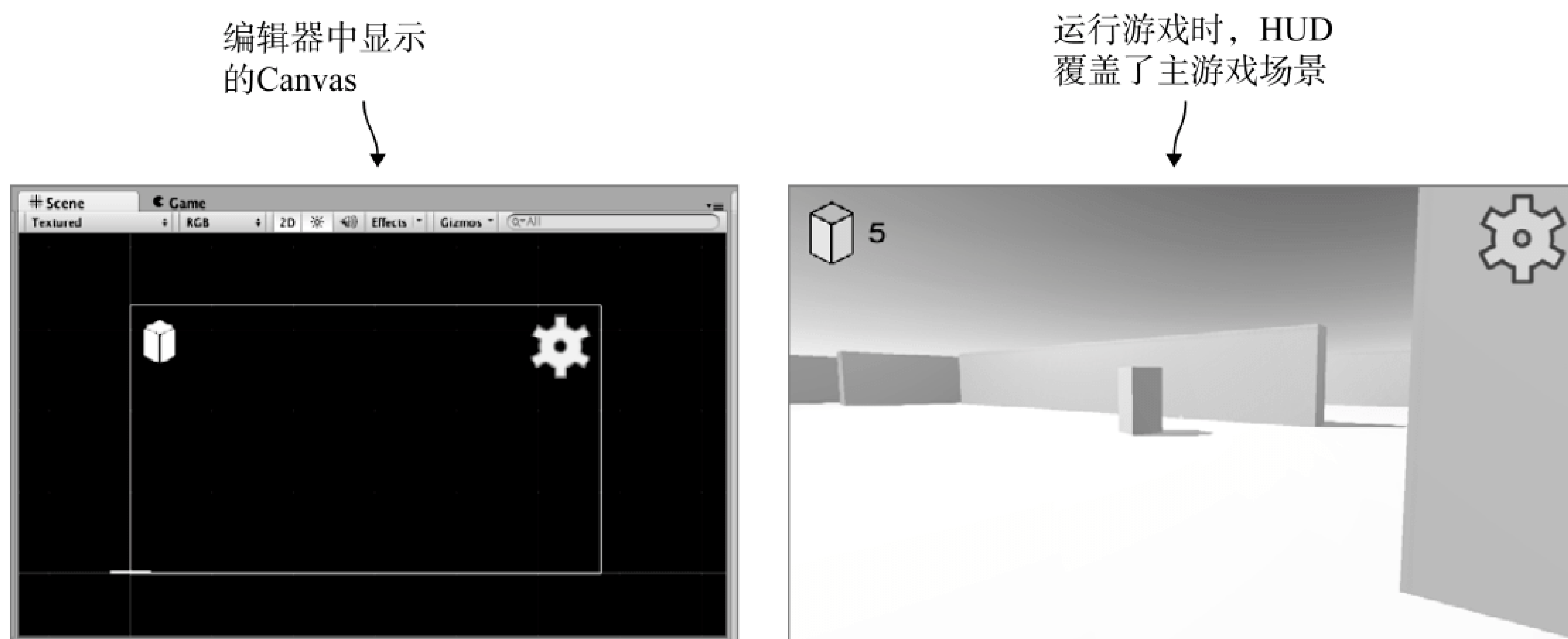


图 7-8　编辑器中看到的 GUI(左图)和运行游戏时看到的 GUI(右图)

前面在 3D 游戏中使用 2D 图像显示了 HUD！还有一个更复杂的可视化设置需要完成：相对于画布来定位 UI 元素。

7.2.3　控制 UI 元素的位置

所有 UI 对象都有锚点(anchor)，在编辑器中显示为 X 形状(如图 7-9 所示)。锚点是一种在 UI 中灵活定位对象的方式。

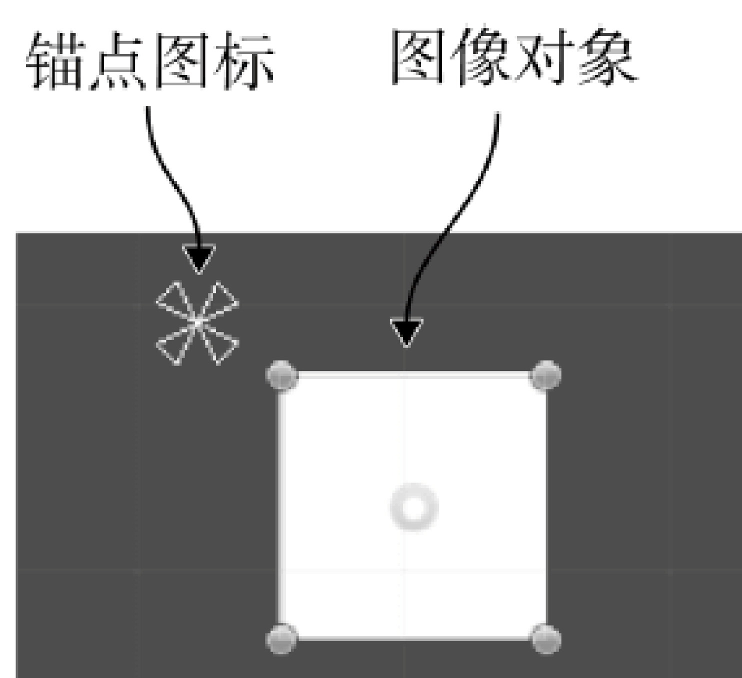


图 7-9　图像对象的锚点

定义　对象的锚点是对象附加到画布或屏幕的点。它决定了计算对象的位置所依赖的点。

位置是类似“X 轴偏移 50 像素”这样的值。但存在一个问题：这 50 像素是相对什么而言？这正是锚点存在的原因。锚点的作用是对象相对于锚点放置，而锚点相对于画布移动。锚点的定义类似于“屏幕中心”，那么当屏幕改变大小时，锚点依然在其中心。类似的，将锚点设置为屏幕的右边，会让对象植根于屏幕的右边，而不管屏幕是否改变大小(例如，假设游戏在不同的显示器上运行)。

理解上述内容最简单的方式就是实践。选择图像对象并观察 Inspector。锚点设置(如图 7-10 所示)会显示在 transform 组件下的右侧。默认情况下，UI 元素把其锚点设置为 Center，但是可以将这个图像的锚点设置为 Top Left；图 7-10 演示了如何使用 Anchor Presets 对锚点进行调整。

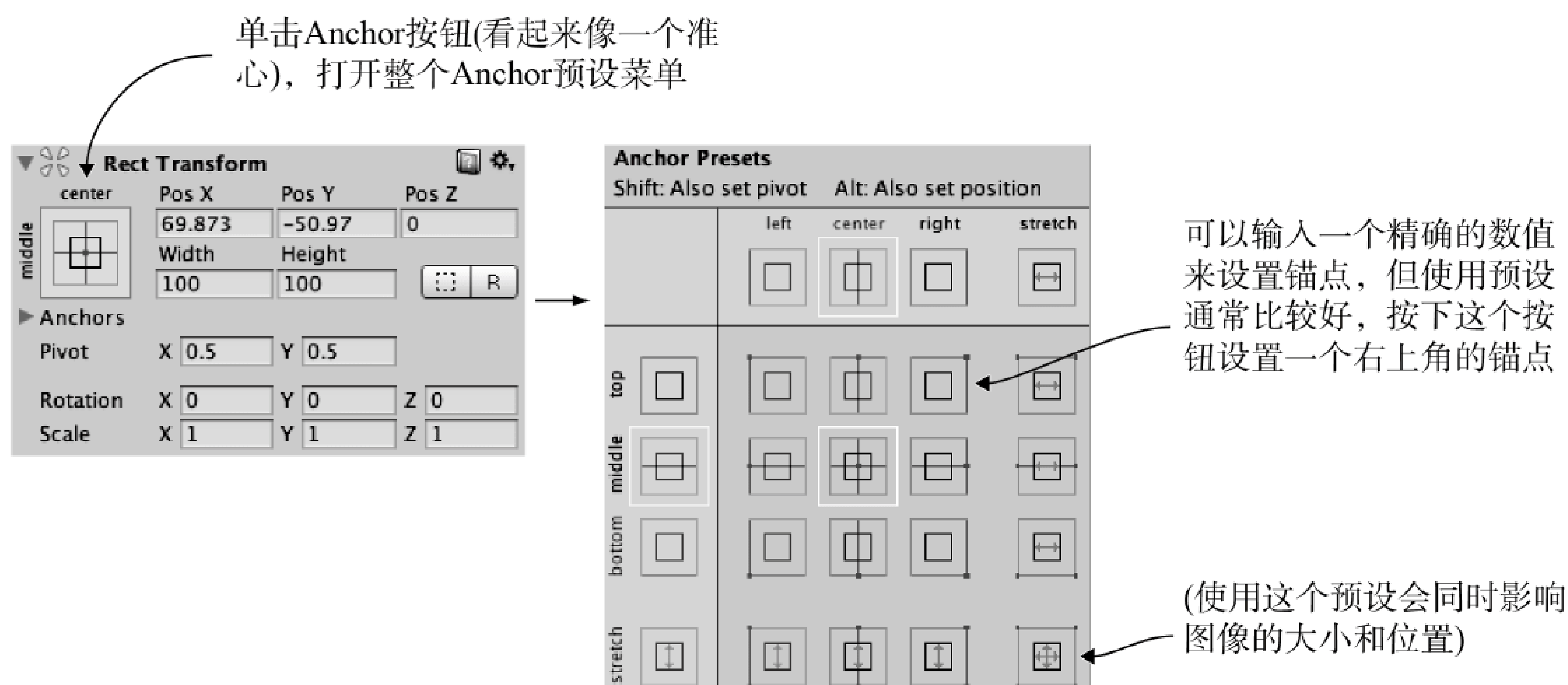


图 7-10　调整锚点设置

同样，下面修改齿轮按钮的锚点，设置这个对象的锚点为 Top Right，单击右上角的 Anchor Preset。现在尝试缩放窗口的左右，单击并拖动 Scene 视图的边。由于存在锚点，UI 对象会在画布改变大小时一直留在其角落位置上。如图 7-11 所示，这些 UI 元素会在屏幕移动时固定在其位置上。

> **提示**　锚点能同时调整比例和位置。本章不打算探讨这些功能，但图像的每个角落都可以定位到屏幕的不同位置。图 7-11 中图像没有改变大小，但可以调整锚点，使图像在屏幕改变大小时进行缩放。

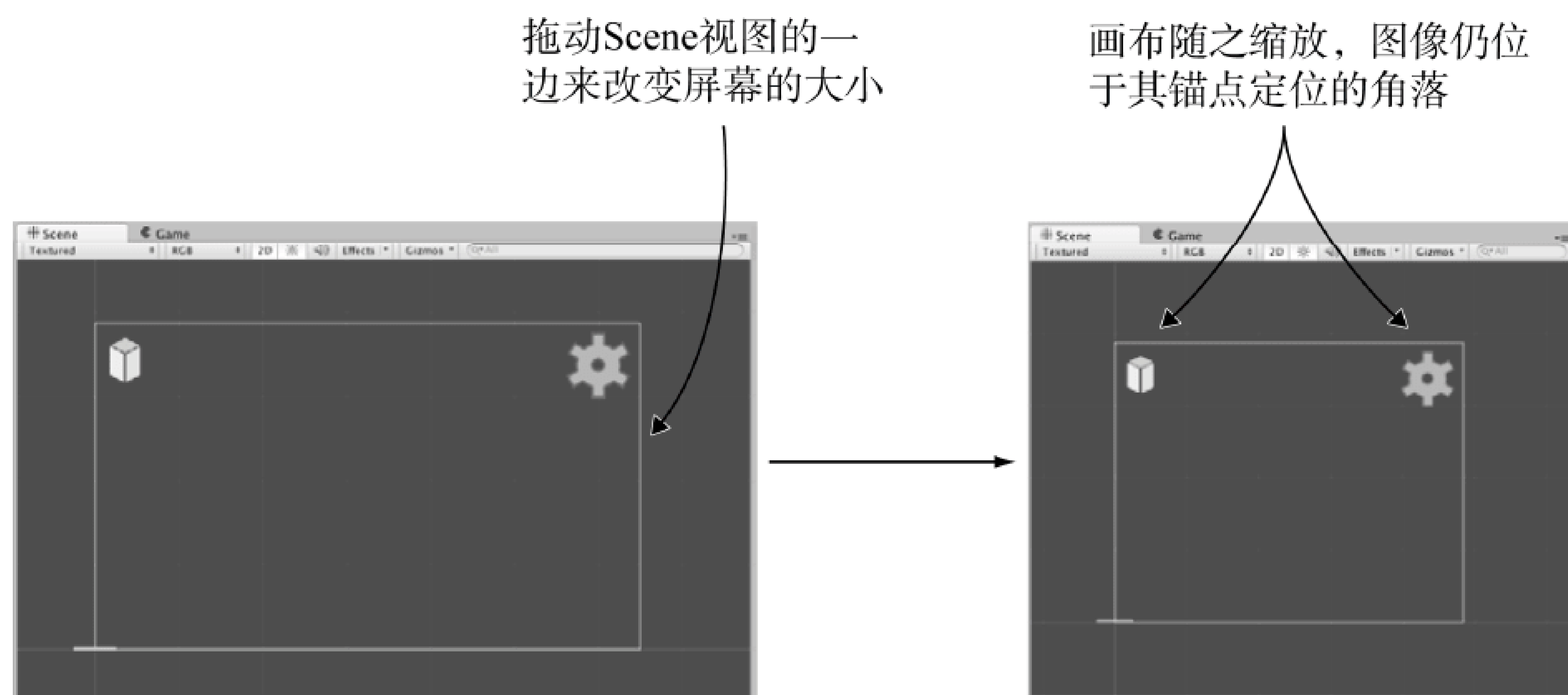


图 7-11　当屏幕改变大小时锚点保持原来的位置

所有的可视化设置已经完成，下面该编写程序进行交互了。

7.3　编写 UI 中的交互

在与 UI 交互之前，需要有鼠标光标。该游戏在 RayShooter 代码的 Start()方法中调整 Cursor 设置。这些设置锁定并隐藏了鼠标光标，这种行为适用于 FPS 游戏的控件，但是影响了 UI 的使用。从 RayShooter.cs 中移除这些设置光标的代码，这样就可以单击 HUD。

只要打开了 RayShooter.cs，就可以确保与 GUI 交互时不能射击。代码清单 7.2 可以实现这一点。

代码清单 7.2　在 RayShooter.cs 中添加 GUI 检查的代码

```
using UnityEngine.EventSystems;    ← 包含 UI 系统代码框架
...
void Update() {                    斜体代码已经在脚本中，
    if (Input.GetMouseButtonDown(0)&&   ← 此处显示仅便于参考
!EventSystem.current.IsPointerOverGameObject()) {    ← 检查 GUI 未被使用
    Vector3 point = new Vector3(
      camera.pixelWidth/2, camera.pixelHeight/2, 0);
    ...
```

现在可以运行游戏并单击按钮，尽管该游戏还没有任何功能。可以看到，当鼠标移到按钮并单击时，它的颜色发生了变化。这个鼠标悬停和单击时表现为默认的染色，对于每个按钮都可以修改这个染色，但现在默认颜色看起来还可以。可以加速默认的淡入淡出行为，Fade Duration 是按钮组件中的设置，可以尝试将其减少为 0.01 并观察

按钮如何变化。

提示　有时，UI 默认的交互控件会影响游戏。记住 EventSystem 对象会同画布一起自动创建。EventSystem 对象控制 UI 交互控件，默认情况下，它使用方向按键来与 GUI 进行交互。需要关闭 EventSystem 中的方向键，避免不小心与 GUI 交互：在 EventSystem 的设置中，不要选中 Send Navigation Event 复选框。

但当单击按钮时什么事情也没有发生，因为现在还没有将按钮关联到任何代码。下面将编写代码。

7.3.1　编写不可见的 UIController

通常，所有 UI 元素的 UI 交互都一样，都以一系列标准步骤进行编写：

(1) 在场景中创建 UI 对象(前一节创建的按钮)

(2) 编写当操作 UI 时调用的脚本

(3) 将脚本附加到场景的对象上

(4) 通过脚本将 UI 元素(如按钮)关联到对象上

为了按照这些步骤进行操作，首先需要创建控制器对象来关联按钮。创建一个 UIController 脚本(如代码清单 7.3 所示)，并将该脚本拖动到场景中的控制器对象上。

代码清单 7.3　对按钮编程的 UIController 脚本

```
using UnityEngine;
using UnityEngine.UI;                  ← 导入 UI 代码框架
using System.Collections;

public class UIController : MonoBehaviour {
  [SerializeField] private Text scoreLabel;    ← 在场景中引用文本对象，设置文本属性

  void Update() {
    scoreLabel.text = Time.realtimeSinceStartup.ToString();
  }

  public void OnOpenSettings() {       ← 由设置按钮调用的方法
    Debug.Log("open settings");
  }
}
```

提示　为什么需要为 SceneController 和 UIController 指定不同的对象？实际上，这个场景很简单，可以用一个控制器来处理 3D 场景和 UI。然而随着游戏越来越复杂，将 3D 场景和 UI 分为不同的模块并在它们之间间接通信，将会越来越有利。通常这个概念能很好地从游戏延伸到软件。软件工程师称这个原理为关注分离(separation of concerns)。

现在将对象拖动到组件槽，连接它们。将分数标签(之前创建的文本对象)拖动到UIController 的文本槽。UIController 中的代码设置了显示在该标签上的文本。当前代码显示了一个定时器，以测试文本显示；在后面会将它修改为分数。

接下来，将一个 OnClick 条目添加到按钮上，并将控制器对象拖动到它上面。选中按钮，观察它在 Inspector 中的设置。其底部有一个 OnClick 面板，初始时该面板为空，但(如图 7-12 所示)可以单击+按钮，在面板上添加一个条目，每个条目都定义了一个函数，当单击按钮时就会调用这个函数。这个列表中包含了一个对象槽和一个用于调用函数的菜单。将控制器对象拖动到对象槽上，并在菜单中查找 UIController，选择其中的 OnOpenSettings()。

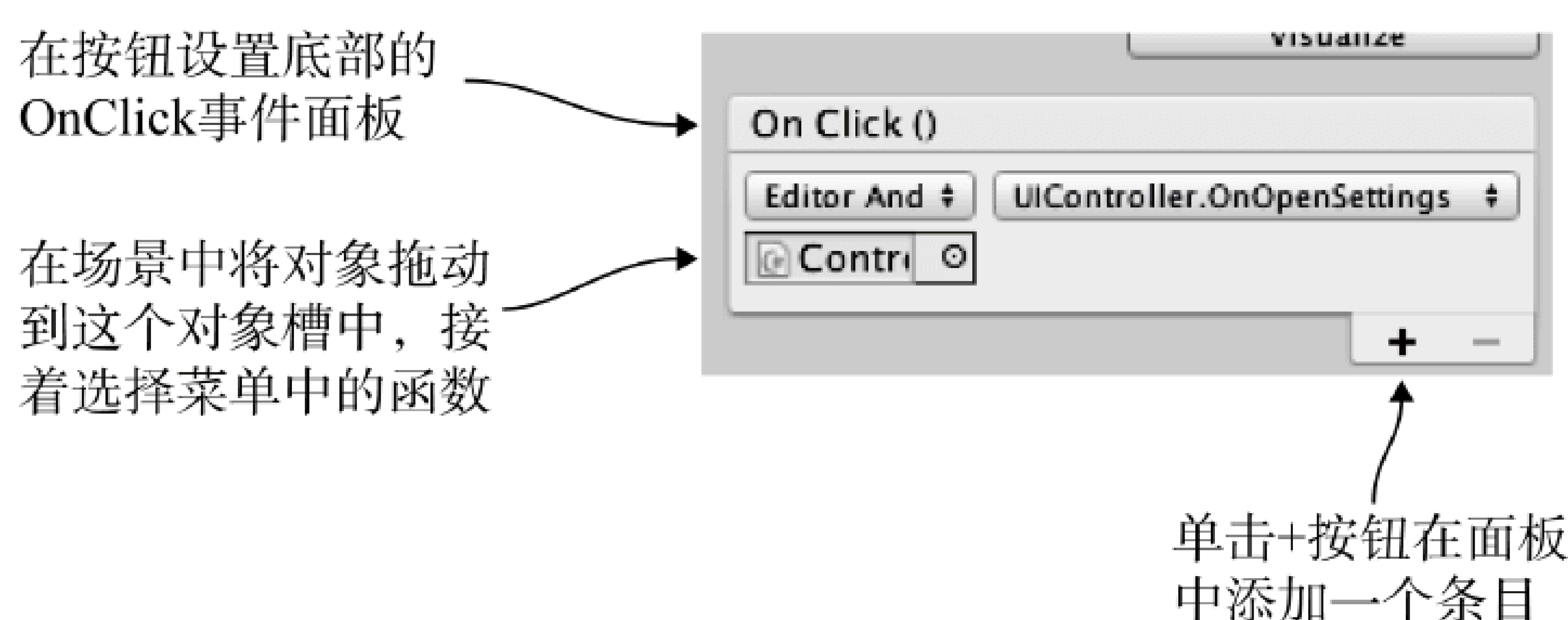


图 7-12 位于按钮设置底部的 OnClick 面板

响应其他鼠标事件

OnClick 是按钮组件显示出来的唯一事件，但 UI 元素能响应各种不同的交互。为了使用默认交互以外的交互，可以使用 EventTrigger 组件。

将一个新组件添加给按钮对象，在组件的菜单中查找 Event 部分。从该菜单中选择 EventTrigger。按钮的 OnClick 只响应一次完整单击(按下鼠标按键并释放)，接下来尝试响应鼠标按下且不松开的事件。执行之前和响应 OnClick 事件一样的操作，只是响应另一个事件。首先将另一个方法添加给 UIController:

```
...
public void OnPointerDown() {
    Debug.Log("pointer down");
}
...
```

现在单击 Add New Event Type，给 EventTrigger 组件添加一个新类型。选择 Pointer Down 作为事件。这个操作将创建空的事件面板，就像 OnClick 面板一样。单击+按钮，添加事件列表，将控制器对象拖动到这个新增事件上，并选择菜单中的 OnPointerDown()。这就完成游戏了。

运行游戏并单击按钮，在控制台中输出调试消息。同样，当前代码只是随机输出，以测试按钮的功能。我们希望打开一个弹出的设置窗口，因此接下来创建弹出窗口。

7.3.2　创建弹出窗口

UI 包含一个用于打开弹出窗口的按钮，但目前还没有弹出窗口。弹出窗口是一个新的图像对象，在这个对象上附加有几个控件(例如，按钮和滑动条)。第一步是创建一个新图像，因此选择 GameObject | UI | Image。和之前一样，新图像在 Inspector 中有称为 Source Image 的图像槽。将精灵拖动到那个槽上，设置这个图像。这次使用称为 popup 的精灵。

通常，缩放精灵以覆盖整个图像对象，这是分数和齿轮图像的工作方式，单击 Set Native Size 按钮，将对象的大小设置为图像对象的大小。这个行为是图像对象的默认设置，但弹出窗口将做一些不同的处理。

如图 7-13 所示，图像组件有 Image Type 设置。这个设置默认为 Simple，这是之前使用的正确图像类型。对于弹出窗口，将 Image Type 设置为 Sliced。

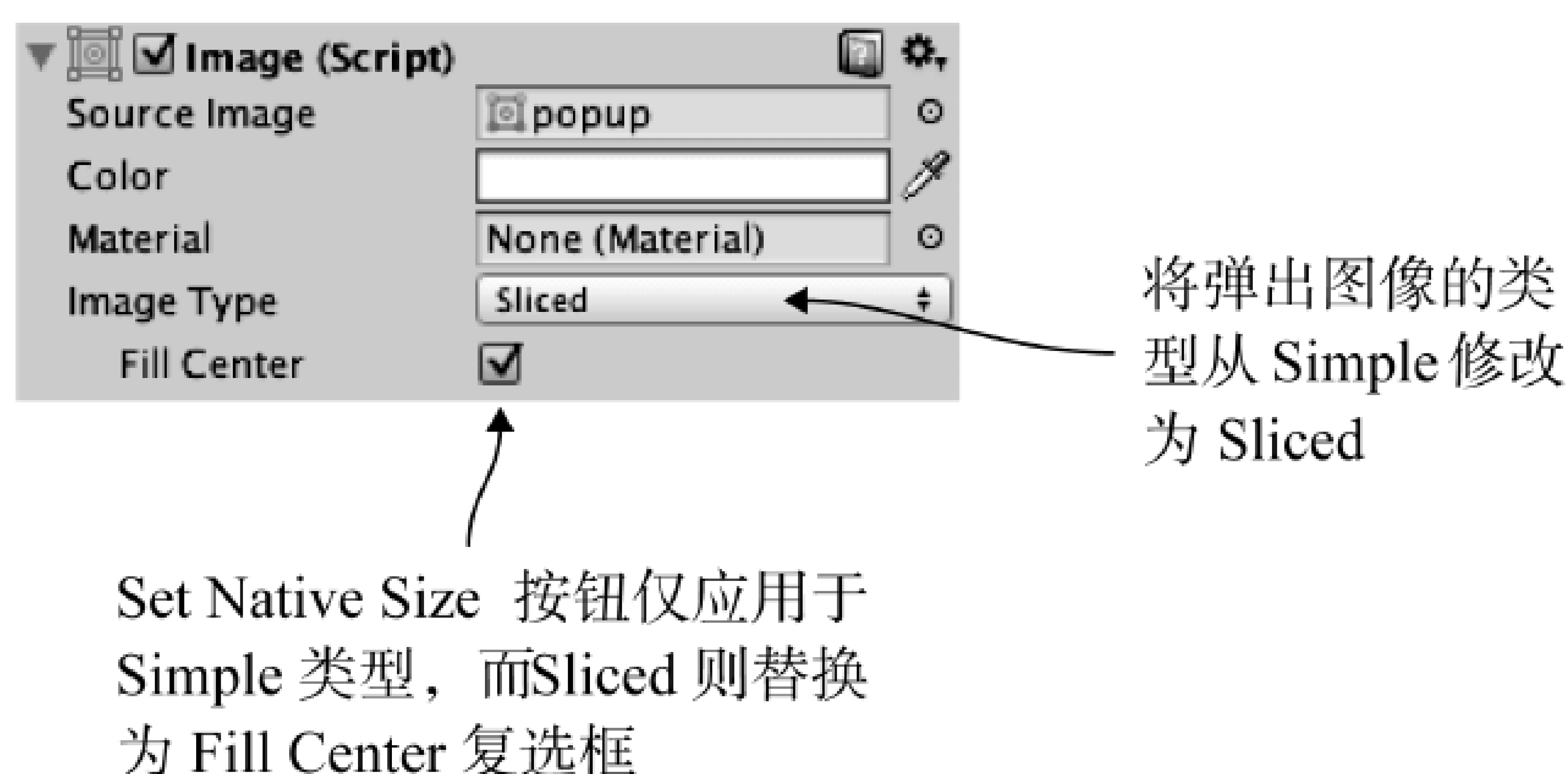


图 7-13　图像组件的设置，包括图像类型

定义　切割图像(sliced image)是把图像切割为九份，各个部分相互独立，分别缩放。通过从中间缩放图像边缘，可以确保图像缩放为任何期望的尺寸，且边缘清晰。在其他开发工具中，此类图像通常在其名称中有个“九”字(例如，九切割、九片、缩放九)，表示图像有九部分。

在切换到切割图像之后，Unity 可能会在组件设置中显示一个错误，表明图像没有边框。这是因为 popup 精灵还没有设置为九部分。为了设置该精灵，首先选择 Project 视图中的 popup 精灵。在 Inspector 中应该会看到 Sprite Editor 按钮(如图 7-14 所示)，单击该按钮，打开 Sprite Editor 窗口。

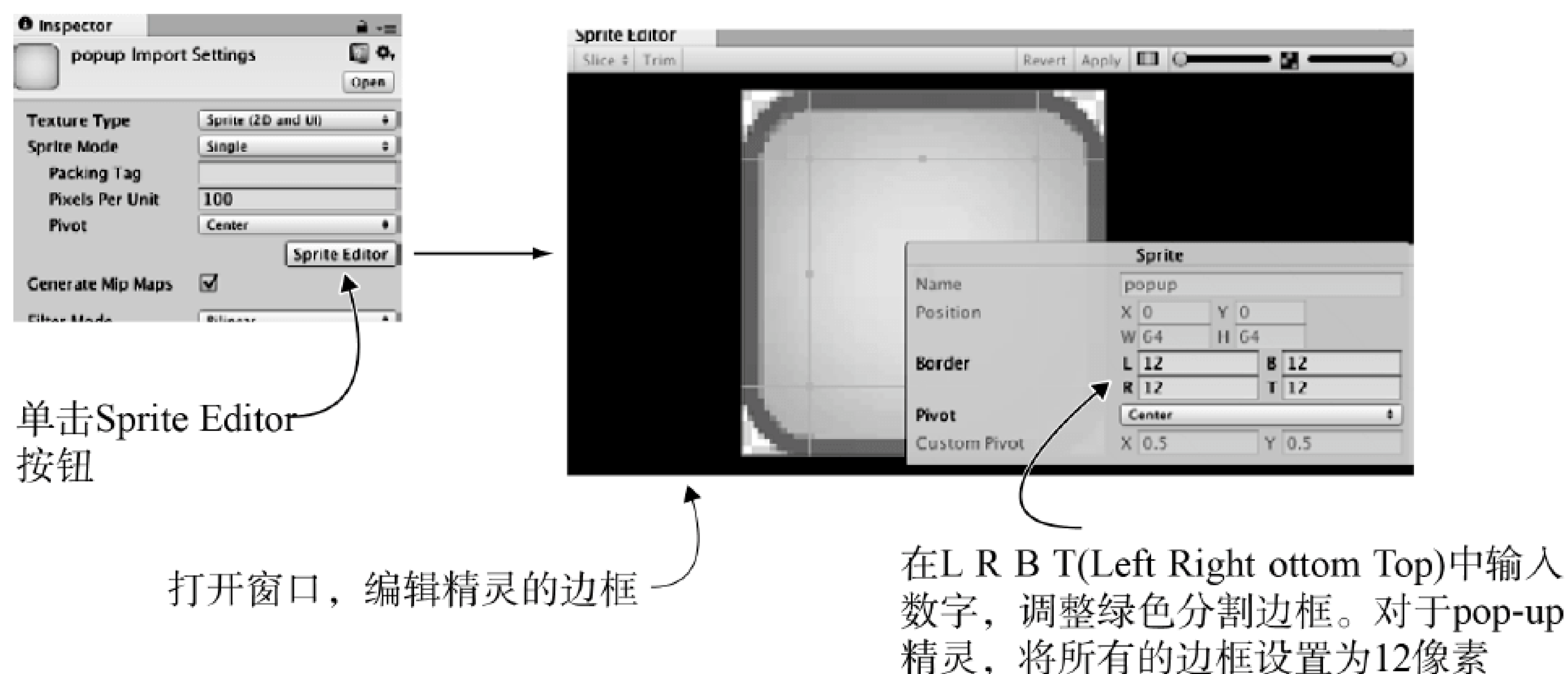


图 7-14　Inspector 中的 Sprite Editor 按钮和弹出窗口

在 Sprite Editor 中可以看到，绿色的线指示了图像是如何切割的。初始时图像不会有任何边框(即，所有 Border 都设置为 0)。增加 4 条边的边框宽度，得到如图 7-14 所示的边框。因为所有 4 条边(左、右、底部、顶部)的边框都设置为 12 像素宽，所以边框线会叠加为九部分。关闭编辑器窗口并应用修改。

现在精灵已定义为九部分，切割的图像将正常工作(Image 组件设置将显示 Fill Center，请确保开启了这个设置)。单击并拖动图像角落的蓝色指示器来缩放它(如果没看到任何缩放指示器，就切换到第 5 章描述的 Rect 工具)。当中心部分缩放时，边框部分的大小将保持不变。

由于边框部分的大小保持不变，因此切割图像可以缩放为任意大小，且边缘仍旧清晰。这非常适合于 UI 元素——不同窗口可能大小不同，但看起来应该一样。对于弹出窗口，设置宽度为 250，高度为 200，如图 7-15 所示同时设置坐标为(0, 0, 0)，让它居中。

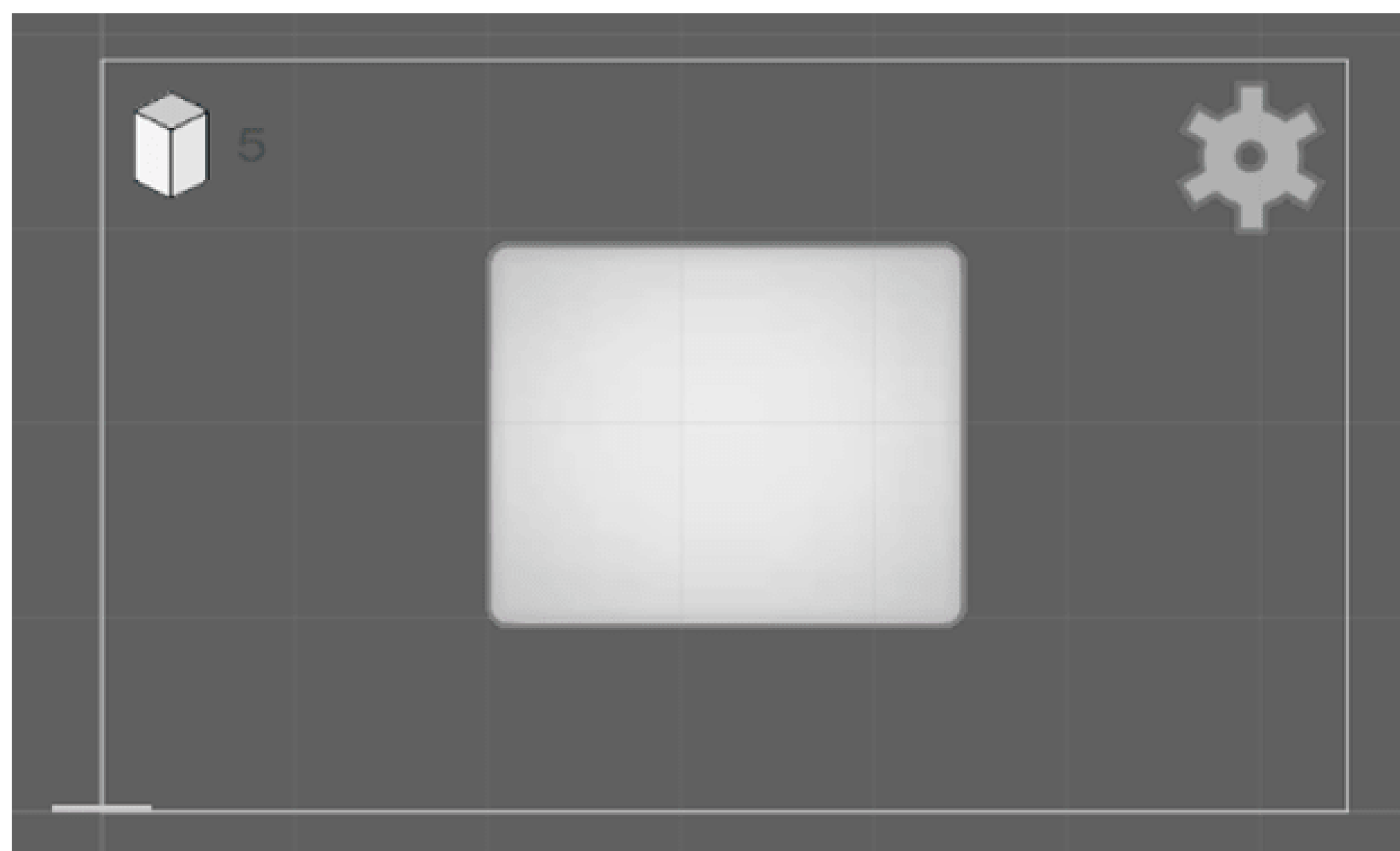


图 7-15　切割图像缩放为 pop-up 对象的大小

提示　UI 图像如何彼此堆叠取决于它们在 Hierarchy 视图中的顺序。在 Hierarchy 列表中，将弹出对象拖动到其他 UI 对象的上面(当然，总是要附加到画布上)。现在在 Scene 视图中移动弹出窗口，图像和弹出窗口就会重叠显示。最后将弹出窗口拖动到画布底部，使它显示在其他任何 UI 元素之上。

弹出对象现在已经设置好，因此可以为它编写代码。创建一个称为 SettingsPopup 的脚本(查看代码清单 7.4)，并将脚本拖动到弹出对象上。

代码清单 7.4　用于 pop-up 对象的 SettingsPopup 脚本

```
using UnityEngine;
using System.Collections;

public class SettingsPopup : MonoBehaviour {
    public void Open() {
        gameObject.SetActive(true);        ←—— 开启对象，打开窗口
    }
    public void Close() {
        gameObject.SetActive(false);       ←—— 使对象无效，关闭窗口
    }
}
```

接下来，打开 UIController.cs 做一些调整，如代码清单 7.5 中所示。

代码清单 7.5　调整 UIController 来处理弹出窗口

```
...
[SerializeField] private SettingsPopup settingsPopup;
void Start() {
    settingsPopup.Close();          ← 游戏开始时关闭弹出窗口
}
...
public void OnOpenSettings() {
    settingsPopup.Open();           ← 使用弹出窗口的方法替换调试文本
}
...
```

这段代码添加了一个弹出对象的对象槽，因此将弹出对象拖动到 UIController 上。当运行游戏时，弹出对象是关闭的，当单击设置按钮时，会打开它。

此时还无法关闭弹出窗口，因此将关闭按钮添加到弹出窗口上。这一步和之前创建按钮的步骤类似：选择 GameObject | UI | Button，将新按钮定位到弹出窗口的右上角，将 close 精灵拖动到这个 UI 元素的 Source Image 属性，接着单击 Set Native Size，正确设置图像的大小。与之前的按钮不同，这次需要文本标签，因此选择文本，并在文本域中输入 Close，并将 Color 设置为白色。在 Hierarchy 视图中，将按钮拖动到弹出对象上，使其成为弹出窗口的子节点。最后打磨其视觉效果，将按钮的 transition 属性的 Fade Duration 值调整为 0.01，使 Normal Color 更暗，设置为(110, 110, 110, 255)。

为了让按钮关闭弹出窗口，需要一个 OnClick 条目。单击 OnClick 面板的+按钮，将弹出窗口拖动到对象槽中，并从函数列表中选择 Close()。现在运行游戏，这个按钮就会关闭弹出窗口。

弹出窗口已添加到 HUD 中。窗口当前还是空白的，因此接下来将一些控件添加到窗口中。

7.3.3　使用滑动条和输入域设置值

将一些控件添加到弹出的设置窗口包括两个步骤，这与之前创建按钮的步骤一样。创建 UI 元素并将其附加到画布上，再将这些对象关联到脚本上。我们需要的输入控件是一个滑动条和一个文本域，还需要一个用于标识滑动条的静态文本标签。选择 GameObject | UI | Text 创建文本对象，选择 GameObject | UI | InputField 创建文本域，而选择 GameObject | UI | Slider 创建滑动条对象(如图 7-16 所示)。

在 Hierarchy 视图中拖动这三个对象，使其成为弹出窗口的子对象，并如图 7-16 所示定位它们，将它们排列在弹出窗口的中心。设置文本为 Speed，将它作为滑动条的标签。输入域用于输入文本，Text 在玩家输入任何内容之前都显示在盒子中；这里设置 Text 的值为 Name。可以让 Content Type 和 Line Type 保留为默认值，如果需要，可以使用 Content Type 来限制输入类型，例如只输入字母或数字；另外，可以使用 LineType 将文本输入切换为单行或多行文本。

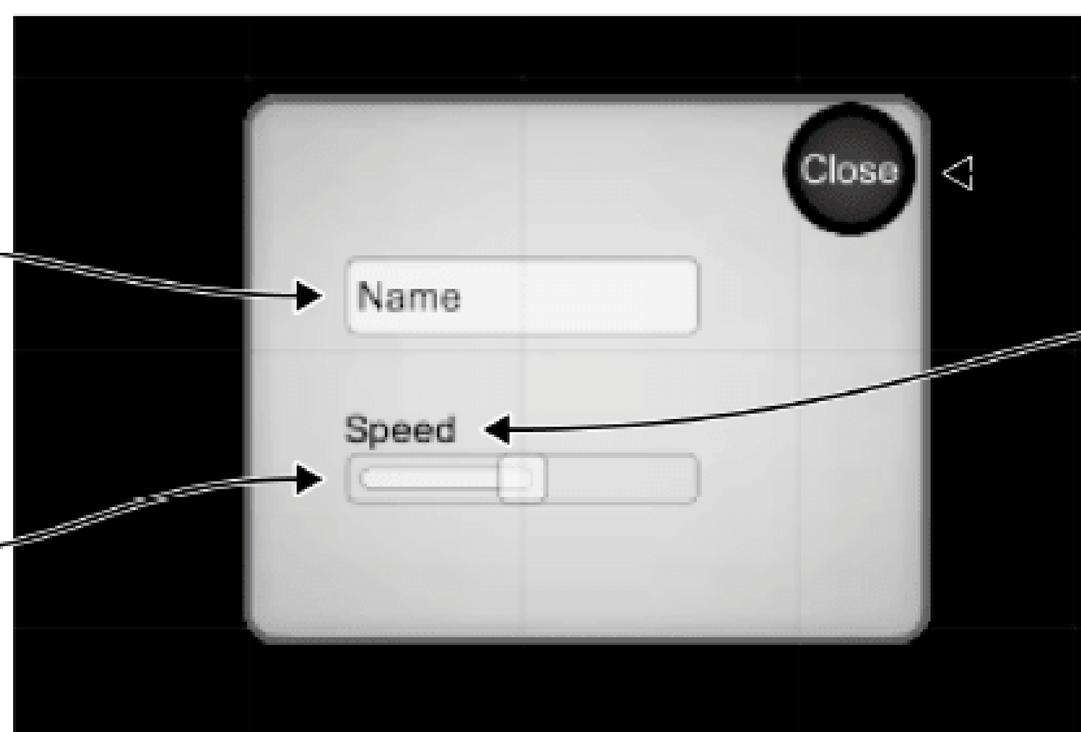


图 7-16　添加到弹出窗口的输入控件

警告　当文本标签覆盖在滑动条上时，将无法单击滑动条。在 Hierarchy 中将文本对象放在滑动条之上，可以确保文本对象出现在滑动条的下方。

警告　在本例中，应该将 Input Field 保留为默认大小，但是如果决定缩小它，那么只减小 Width，不减小 Height。如果将 Height 设置为小于 30 就太小了，无法显示文本。

就其滑动条自身而言，组件 Inspector 的底部有很多设置。Min Value 默认设置为 0，保持其默认设置。Max Value 默认为 1，但本示例中需要修改为 2。类似的，Value 和 Whole Numbers 都可以保留其默认设置，Value 控制滑动条的开始值，而 Whole Numbers 将它限制为 0 1 2 而不是小数值(本例不需要这个限制)。

现在所有对象都已处理完毕。现在需要编写关联对象的代码，如代码清单 7.6 所示，将一些方法添加到 SettingsPopup.cs 中。

代码清单 7.6　SettingsPopup 中用于弹出窗口的输入控件的方法

```
...
public void OnSubmitName(string name) {     ← 当用户在输入域输入时触发该方法
    Debug.Log(name);
}
public void OnSpeedValue(float speed) {     ← 当用户调整滑动条时触发该方法
    Debug.Log("Speed: " + speed);
}
...
```

很好，现在有了用于控件的方法。下面开始处理输入域，在输入域的设置中可以看到 End Edit 面板，其中列出的事件会在完成输入时触发。给这个面板添加一个条目，将弹出窗口拖动到对象槽，并在函数列表中选择 OnSubmitName()。

警告　一定要在 End Edit 面板顶部的 Dynamic String 部分选择该函数，而不是在底部的 Static Parameters 部分选择。OnSubmitName()方法会出现在这两部分，但在 Static Parameters 中选择，将只发送提前定义的单个字符串，而在 Dynamic String 中选择时发送的是输入域中输入的任何内容。

对于滑动条完成相同的步骤：查找组件设置底部的事件面板(本示例中是 OnValueChanged)，单击+按钮添加一个条目，将滑动条拖动到对象槽上，并在列出的动态值函数中选择 OnSpeedValue()。

现在所有输入控件已经关联到弹出脚本中的代码。运行游戏、移动滑动条或者在输入之后按下 Enter 键并观察控制台。

使用 PlayerPrefs 保存游戏过程的设置

Unity 中有一些不同的方法用于保存持久化的数据，最简单的方法称为 PlayerPrefs。Unity 提供了一种抽象方式(也就是说不必关心细节)，可以保存用于所有平台(使用它们不同的文件系统)的少量信息。PlayerPrefs 不适合保存大量数据(后续章节将使用其他方法来保存游戏进度)，但它们适用于保存游戏设置。

PlayerPrefs 提供了一些简单的命令用于获取和设置值(它的原理类似哈希表或字典)。例如，在 SettingsPopup 脚本的 OnSpeedValue()方法内，添加代码行 PlayerPrefs. SetFloat("speed", speed); 可以保存速度设置。OnSpeedValue 方法将浮点数保存到 speed 值中。

类似的，可以将滑动条初始化为所保存的值。将如下代码添加到 SettingsPopup 脚本中：

```
using UnityEngine.UI;
...
[SerializedField]private Slider speedSlider;
void Start() {
   speedSlider.value = PlayerPrefs.GetFloat("speed", 1);
}
...
```

注意，get 命令获取值的同时也指定了默认值，以防备之前没有保存 speed 值。

尽管控件生成了调试输出，但它们依然不能真正影响游戏。使 HUD 影响游戏(反之亦然)是本章最后一节要讨论的主题。

7.4　通过响应事件更新游戏

截至目前，HUD 和主游戏之间互不相干，但它们之间应该是相互通信的。为此，可以通过脚本引用来完成，这与其他类型的对象通信所创建的脚本一样，但这样做存在一些缺陷。特别是，这样做将把场景和 HUD 紧密耦合在一起，但它们应相对独立，以便在编辑游戏时不必担心是否破坏 HUD。

为了了解场景中 UI 的行为，要使用消息广播系统。图 7-17 阐述了这个事件消息系统的工作原理：脚本可以注册为侦听事件，其他代码可以广播事件，接着侦听器将被通知有关广播的消息。接下来介绍消息系统以便完成它。

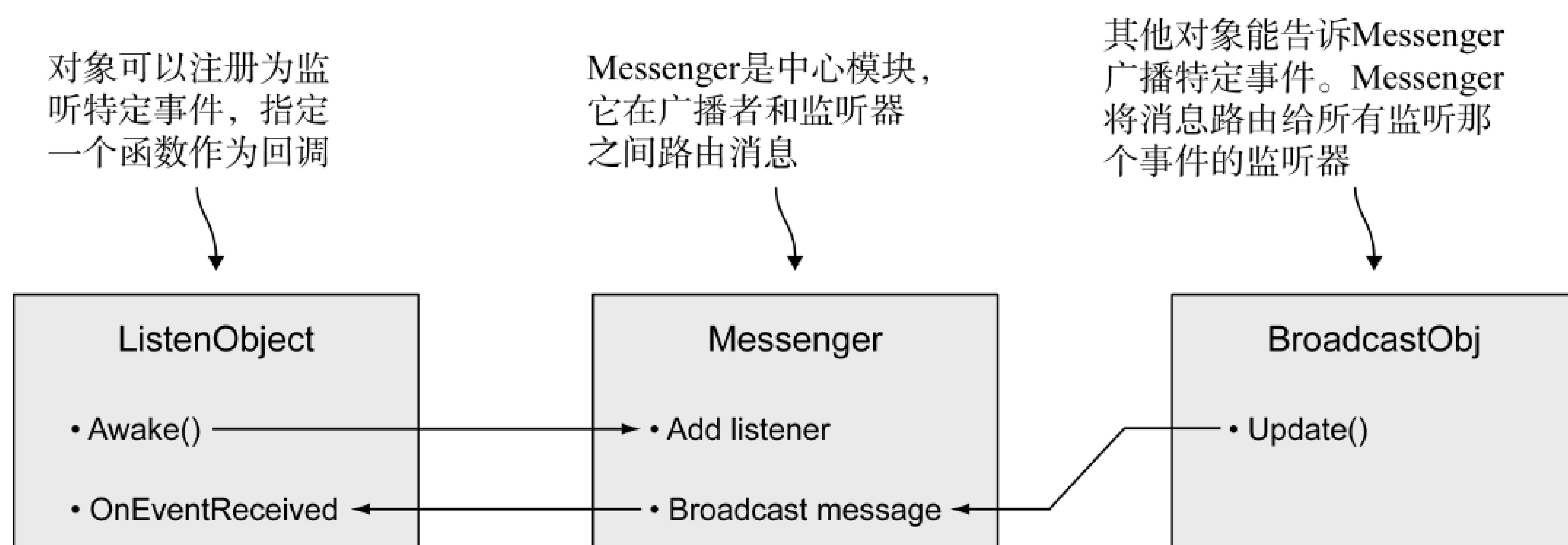


图 7-17　将实现的广播事件系统图

提示 C#有一个内置的系统用于处理事件，为什么不使用它？内置的事件系统要求消息是有目标的，而我们需要的是广播消息系统。目标系统需要代码精确知道消息的来源，而广播的来源可以是任意的。

7.4.1　集成事件系统

为了了解场景中 UI 的行为，需要使用广播消息系统。尽管 Unity 没有内置这个特性，但已经为此下载一个代码库。在附录 D 的资源列表中，有 Unity 的社区 wiki，这是一个由其他开发者提供的免费代码库。它们的消息系统非常适合提供一种与程序其余部分通信的分离方式。当一些代码广播消息时，代码不需要知道关于侦听器的任何消息，在切换或添加对象时，这提供了巨大的灵活性。

创建一个脚本，命名为 Messenger，并将 http://wiki.unity3d.com/index.php/CSharp-Messenger_Extended 页面的代码粘贴到脚本中。接着需要创建一个名为 GameEvent 的脚本，如代码清单 7.7 所示。

代码清单 7.7　Messenger 中使用的 GameEvent 脚本

```
public static class GameEvent {
   public const string ENEMY_HIT = "ENEMY_HIT";
   public const string SPEED_CHANGED = "SPEED_CHANGED";
}
```

代码清单中的脚本为一些事件消息定义了一个常量，消息通常以这种方式组织，不必在每个地方都记住和输入消息字符串。

现在事件消息系统已经准备就绪，接下来开始使用它。首先将它用于从场景到 HUD 的通信，接着用于从 HUD 到场景的通信。

7.4.2　从场景中广播和侦听事件

截至现在，分数依然把显示一个计时器作为文本显示功能的测试。但我们需要显

示击中敌人的计数器，因此修改 UIController 中的代码。首先删除整个 Update()方法，因为这是测试代码。当敌人死亡时，将会触发事件，因此代码清单 7.8 让 UIController 侦听该事件。

代码清单 7.8 将事件侦听器添加到 UIController

```
...
private int _score;

void Awake() {
    Messenger.AddListener(GameEvent.ENEMY_HIT, OnEnemyHit);
}
void OnDestroy() {
    Messenger.RemoveListener(GameEvent.ENEMY_HIT, OnEnemyHit);
}

void Start() {
    _score = 0;
    scoreLabel.text = _score.ToString();

    settingsPopup.Close();
}

private void OnEnemyHit() {
    _score += 1;
    scoreLabel.text = _score.ToString();
}
...
```

声明响应事件 ENEMY_HIT 的方法

当对象被销毁时，清除侦听器，以防止出错

将分数初始化为 0

响应事件时递增分数

首先注意 Awake()和 OnDestroy()方法。就像 Start()和 Update()方法一样，在对象被唤醒或移除时，每个 MonoBehaviour 都会自动响应这两个方法。在 Awake()中添加侦听器，在 OnDestroy()中移除它。这个侦听器是广播消息系统的一部分，当收到消息时，它会调用 OnEnemyHit()。OnEnemyHit()将递增分数，并把值放到分数显示中。

事件侦听器已经在 UI 代码中设置，因此现在不管敌人在何时被击中都需要广播。响应击中敌人的代码位于 RayShooter.cs 中，因此代码清单 7.9 将触发消息。

代码清单 7.9 由 RayShooter 广播事件消息

```
...
if (target != null) {
    target.ReactToHit();
    Messenger.Broadcast(GameEvent.ENEMY_HIT);
} else {
...
```

响应受击时添加的消息广播

在添加消息后运行游戏，观察当击中敌人时分数的显示。每次击中敌人时，分数都会增加。这个示例介绍了从 3D 游戏向 2D 界面发送消息，但我们还需要一个从 2D 界面向 3D 游戏发送消息的示例。

7.4.3　从 HUD 广播和侦听事件

在上一节中，事件从场景广播，被 HUD 接收。同样，UI 控件可以广播玩家和敌人侦听的消息。通过这种方式，设置弹出窗口可以影响游戏设置。打开 WanderingAI.cs 并添加代码清单 7.10 中的代码。

代码清单 7.10　添加到 WanderingAI 的事件侦听器

```
...
public const float baseSpeed = 3.0f;        ← 由速度设置调整的基本速度
...
void Awake() {
    Messenger<float>.AddListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
void OnDestroy() {
    Messenger<float>.RemoveListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
...
private void OnSpeedChanged(float value) {  ← 声明该方法，用于侦听事件 SPEED_CHANGED
    speed = baseSpeed * value;
}
...
```

这里的 Awake()和 OnDestroy()方法也分别用于添加并移除事件侦听器，但在此它们都有值，用于设置 AI 的行走速度。

提示　上一节中的代码使用的只是一般事件，但该消息系统可以传递值和消息。支持侦听器中的值就像添加类型定义一样简单。注意<float>添加到侦听器命令。

现在在 FPSInput.cs 中进行同样的修改，来影响玩家的速度。代码清单 7.11 中的大部分代码和代码清单 7.10 中的一样，只是玩家的 baseSpeed 不同。

代码清单 7.11　添加到 FPSInput 的事件侦听器

```
...
public const float baseSpeed = 6.0f;    ← 这个值相对代码清单 7.10 做了修改
...
void Awake() {
    Messenger<float>.AddListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
void OnDestroy() {
    Messenger<float>.RemoveListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
...
private void OnSpeedChanged(float value) {
    speed = baseSpeed * value;
}
...
```

最后，从 SettingsPopup 中广播速度值，响应滑动条，如代码清单 7.12 所示。

代码清单 7.12　从 SettingsPopup 中广播消息

把滑动条的值作为<float>事件发送

```
public void OnSpeedValue(float speed) {
   Messenger<float>.Broadcast(GameEvent.SPEED_CHANGED, speed);
   ...
```

现在，调整滑动条时，玩家和敌人的速度都会改变。单击 Play 试试！

练习：修改所生成的敌人的速度

当前只更新已存在于场景中的敌人的速度值，而不会影响新生成的敌人的速度值，新敌人并没有以正确的速度值设置创建。这里将如何设置新生成的敌人的速度值作为练习留给读者。提示：将 SPEED_CHANGED 侦听器添加给 SceneController，因为该脚本生成了敌人。

现在知道如何使用 Unity 提供的新 UI 工具构建图形界面。本章介绍的内容在以后所有的项目中都十分有用，甚至是探讨各种不同的游戏类型时也很有用。

7.5　小结

- Unity 有立即模式的 GUI 系统，也有基于 2D 精灵的新 GUI 系统。
- 将 2D 精灵用于 GUI 需要场景有一个画布对象。
- UI 元素能锚定在可调整画布的相对位置上。
- 设置 Active 属性来打开或关闭 UI 元素。
- 独立的消息传送系统适合于在界面和场景之间广播事件。

第 8 章

创建第三人称 3D 游戏：玩家移动和动画

本章涵盖：

- 给场景添加实时阴影
- 使摄像机环绕它的目标
- 使用线性插值(Lerp)算法平滑修改旋转
- 为跳跃、悬崖、斜坡处理地面检测
- 为逼真的角色应用和控制动画

本章将创建另一个 3D 游戏，但这一次将制作新的游戏类型。第 2 章为第一人称游戏构建了一个移动示例。现在需要编写另一个移动示例，但这次涉及第三人称的移动。最重要的区别是摄像机相对于玩家放置：在第一人称视角中，玩家通过角色来观察周围，而在第三人称视角中，摄像机置在角色的外部。冒险游戏常常使用这种视角，如历史悠久的 *Legend of Zelda* 系列游戏或者 *Uncharted* 系列游戏(如果想了解第一人称视角和第三人称视角的区别，可以参考图 8-3)。

本章的项目是本书中构建的视觉效果很棒的原型之一。图 8-1 展示了如何构建场景。将图 8-1 与第 2 章构建的第一人称场景(见图 2-2)相比较。

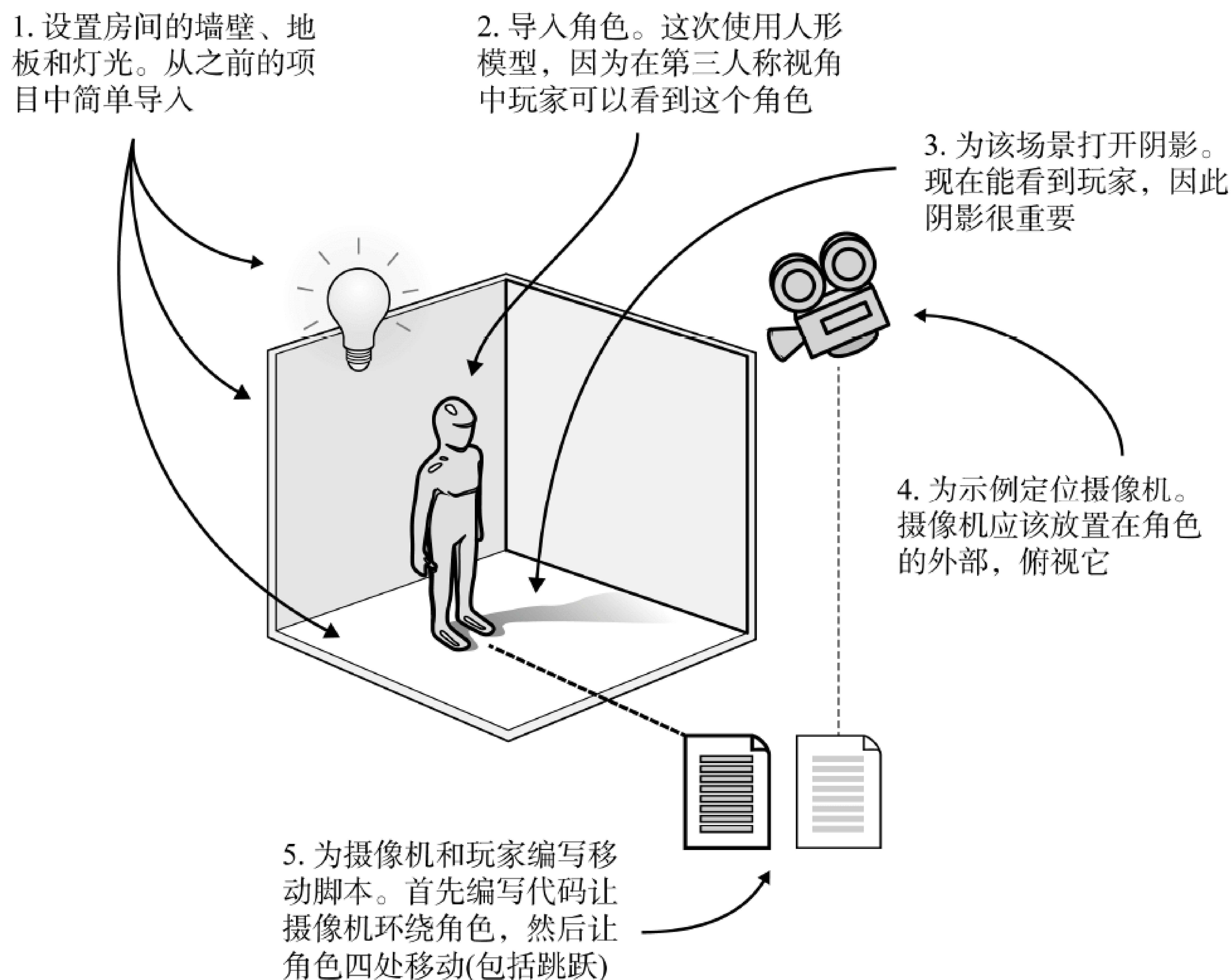


图 8-1　第三人称移动示例的路线图

可以看到，房间的构造是相同的，脚本的用法也大多相同，但玩家的外观、摄像机的位置在不同情况下是不同的。另外，第三人称视角是指，摄像机放在玩家角色的外部，并俯视这个角色。接下来使用人形角色模型(而不是原始的胶囊体)，因为现在玩家可以看到自己。

回想一下，第 4 章讨论的两种美术资源类型是 3D 模型和动画。如前面的章节所述，3D 模型这个词和网格对象是同义词，它是由顶点和多边形定义的静态形状(即网格几何体)。对于人物角色，这个网格几何体塑造成头、胳膊、腿等(见图 8-2)。

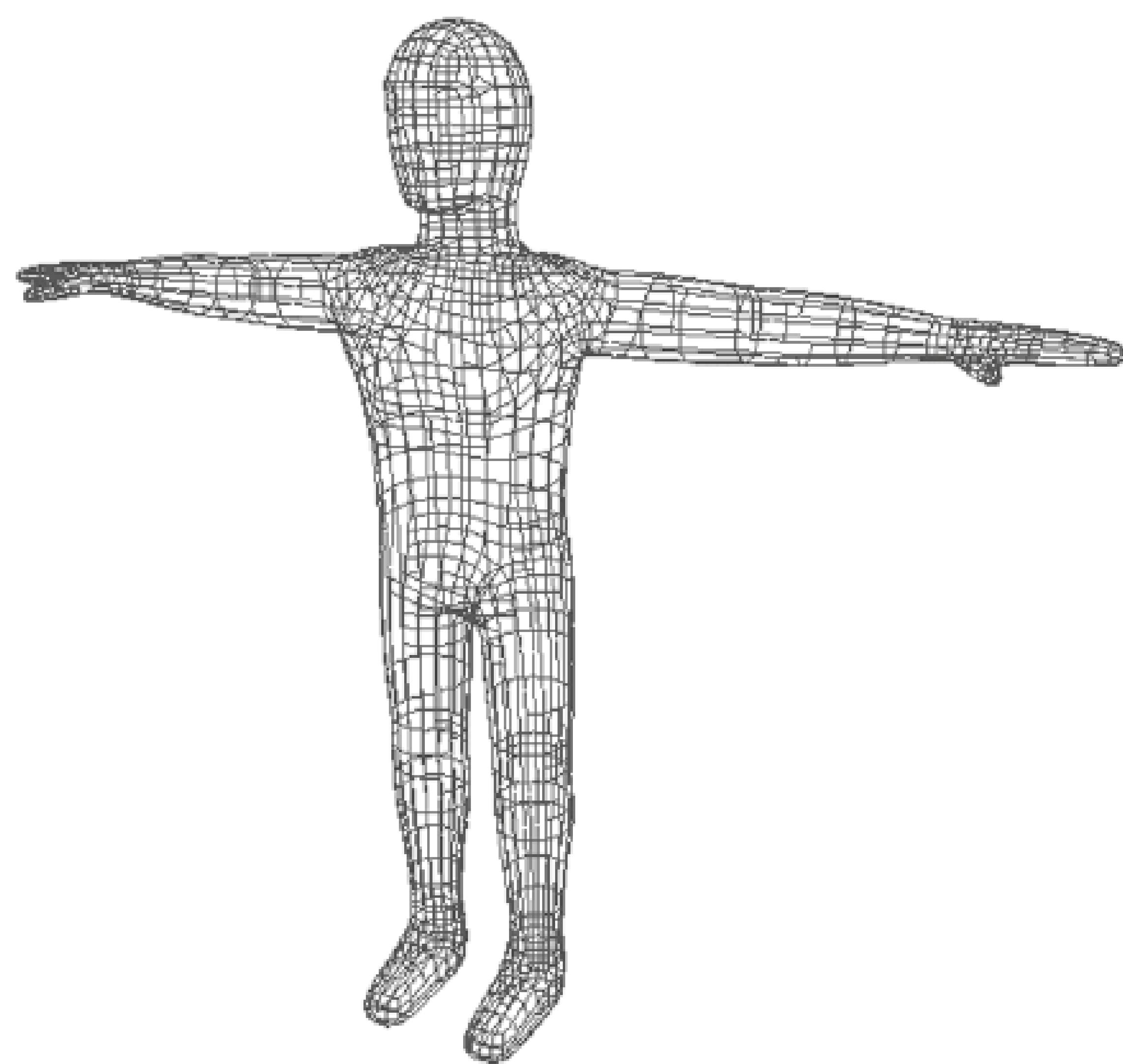


图 8-2　本章所用模型的线框视图

像往常一样，我们将关注路线图的最后一步：编写程序控制场景中的对象。以下是行动计划的回顾：

(1) 将角色模型导入到场景
(2) 实现摄像机控制，以观察角色
(3) 编写脚本，让玩家能够在地面上跑
(4) 给移动脚本添加跳跃功能
(5) 基于移动播放模型的动画

复制第 2 章的项目并修改它，或者创建一个新的 Unity 项目(确保项目设置为 3D 模式，而不是第 5 章的 2D 模式)，复制第 2 章中项目的场景文件。不管采用哪种方式，都可以从本章下载的临时文件夹中获取接下来将使用的角色模型。

注意　我们将基于第 2 章的围墙区域构建本章的项目，墙壁和灯光不变，但替换掉玩家和所有脚本。如果需要它们，可以从第 2 章下载示例文件。

假设从第 2 章已完成的项目(移动的示例)开始，接下来删除本章不需要的内容。首先在 Hierarchy 列表中把摄像机从玩家中分离出来(将摄像机对象从玩家对象上拖出来)。现在删除玩家对象，如果不先分离出摄像机，它也会被删除，但这里只需要删除玩家胶囊，留下摄像机。或者，如果不小心删除了摄像机，就选择 GameObject | Camera，创建一个新的摄像机对象。

同样，删除所有的脚本(包括删除摄像机上的脚本组件以及 Project 视图中的文件)，只剩下墙壁、地板和灯光。

8.1　将摄像机视图调整为第三人称视角

在编写代码让玩家到处移动之前，需要先将一个角色导入场景，并设置摄像机来观察角色。我们将导入一个无脸的人形模型作为玩家角色，然后在上方放置摄像机，使摄像机向下倾斜一个角度来观察玩家。图 8-3 对比了场景在第一人称视角下与在第三人称视角下的外观(有几大块，我们将在本章添加进去)。

场景准备好了，现在要在场景中加入一个角色模型。

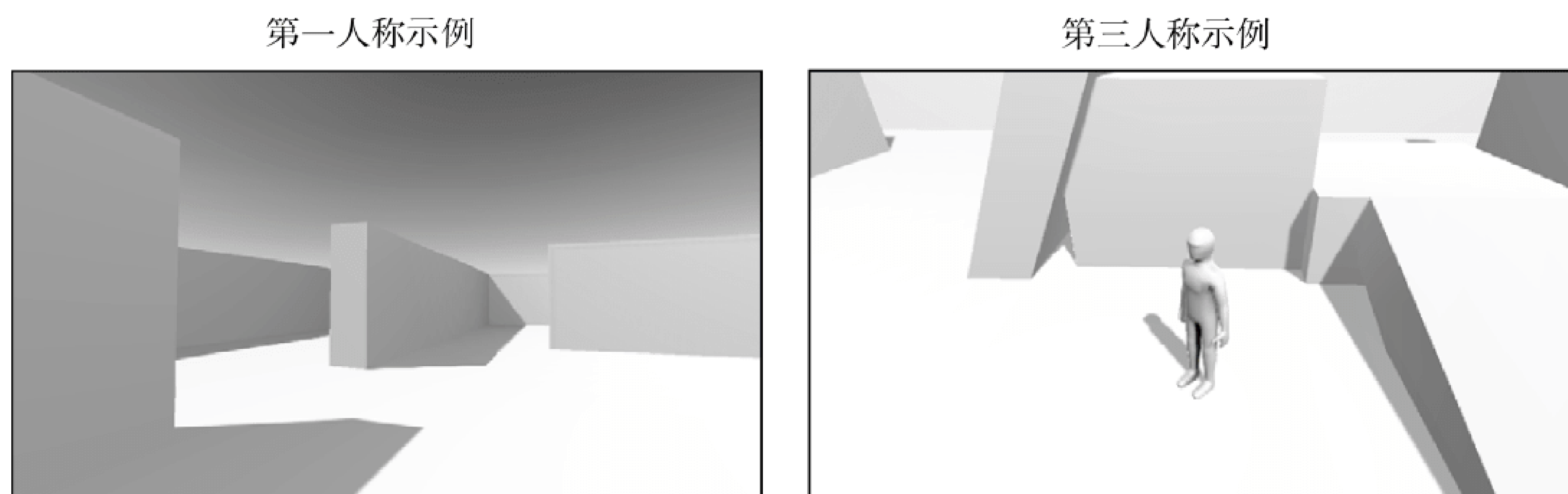


图 8-3　并排比较第一人称视角和第三人称视角

8.1.1　导入一个用于观察的角色

本章下载的临时文件中包括了模型和贴图。在第 4 章，FBX 是模型而 TGA 是贴图。要将 FBX 文件导入到项目中，将该文件拖动到 Project 视图，或者右击 Project 视图，并选择 Import New Asset。然后在 Inspector 中调整模型的导入设置。本章后面将

调整导入的动画，但是现在只需要在 Model 选项卡上做一些调整。首先，将 Scale Factor 的值修改为 10(为了部分抵消 File Scale 值为 0.01 的作用)，使模型大小合适。

在 Scale Factor 设置的下面是 Normals 选项(见图 8-4)。这个设置控制了光线、阴影在模型上的显示，使用了一个称为法线的 3D 数学概念。

定义 法线(Normals)是垂直于多边形的方向向量，它将多边形的朝向告诉计算机。这个朝向用于光照计算。

Normals 的默认设置是 Import，它使用定义在导入网格几何体的法线。但是这个模型没有正确地定义法线，所以对灯光的反应有些奇怪。相反，将 Normals 设置修改为 Calculate，以便 Unity 为每个多边形计算方向向量。

当修改完这两个设置后，单击 Inspector 上的 Apply 按钮。接着将 TGA 文件导入到项目，并将这张图片指定为材质的贴图。在 Materials 文件夹中选择 player 材质。将贴图图像拖到 Inspector 上空的贴图槽。应用贴图之后，模型颜色就不会发生巨大的变化(贴图图像大多是白色)。但在贴图中绘制了阴影，将改善模型的外观。

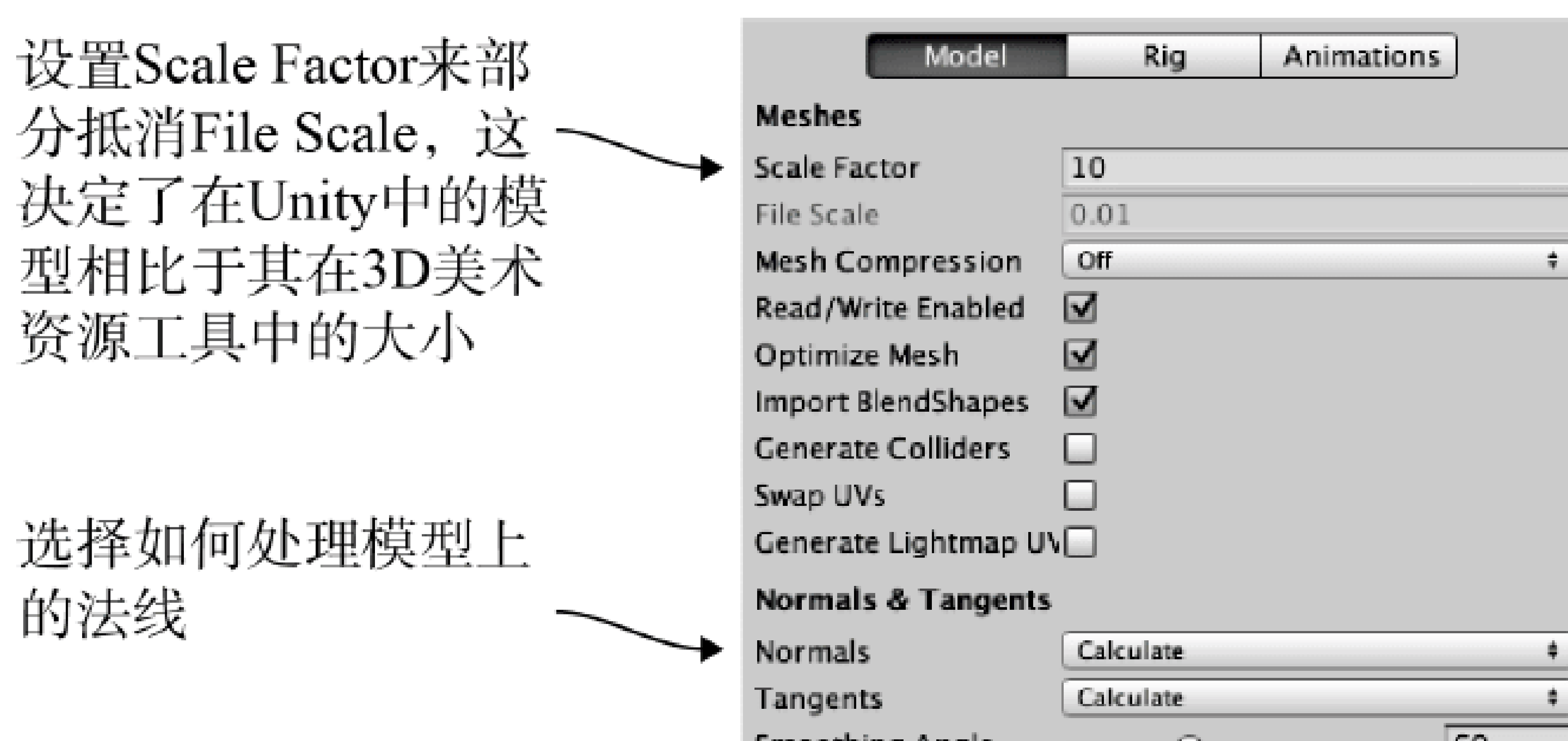


图 8-4　角色模型的导入设置

应用贴图后，把玩家模型从 Project 视图拖到场景中，设置角色位置为(0, 1.1, 0)，以便玩家位于房间的中心，站在地板上。现在场景中有一个第三人称的角色。

注意 这个导入的角色双臂侧平举，而不是垂下双臂的自然姿态。这是因为还没有应用动画，双臂侧平举的姿态称为 T 形姿态。给角色制作动画前，角色的标准默认姿态是 T 形姿态。

8.1.2　将阴影添加到场景

在继续工作之前，先解释一下角色投射的阴影。我们在真实世界里是有阴影的，但在虚拟的游戏世界中却不一定有。很幸运 Unity 能处理这个细节，给新场景自带的默认灯光打开阴影。选择场景中的平行光，然后在 Inspector 中找到 Shadow Type 选项。对于默认的灯光，该设置开启了 Soft Shadows(如图 8-5 所示)。但注意菜单中也有一个 No Shadows 选项。

这就是在这个项目中创建阴影所需的操作，但游戏中的阴影还有很多知识需要了解。计算场景中的阴影是计算机图形学中特别耗时的一部分。所以游戏往往以不同的方式偷工减料，以达到所需的视觉外观要求。这种角色投影被称为实时阴影，因为阴影的计算是在游戏运行时完成的，且跟随移动的对象一起移动。一个很真实的灯光设置会让所有的对象在实时接收和投射光线，但为了使阴影计算运行得足够快，实时阴影受限于阴影的外观或者哪个灯光可以投射阴影。注意，在这个场景中只有平行光投射阴影。

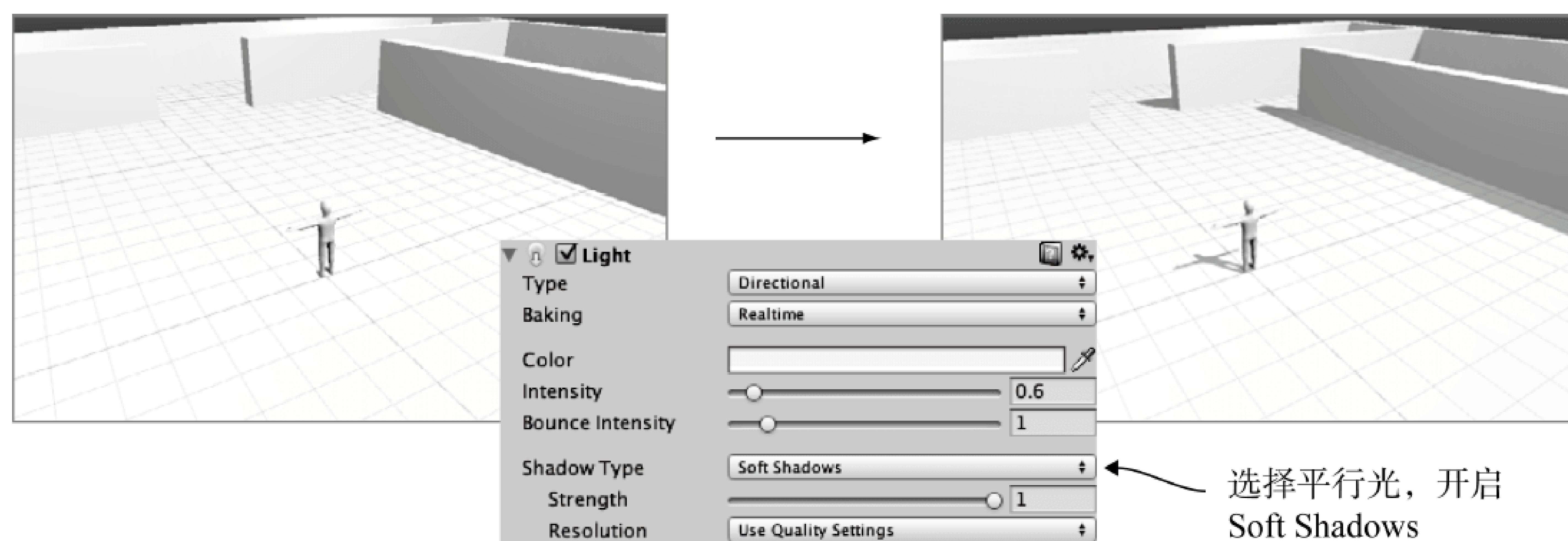


图 8-5　平行光投影之前和投影之后

在游戏中处理阴影的另一种常见方式是使用一种称为光照贴图(lightmapping)的技术。

定义　lightmaps 是应用到关卡几何体的贴图，这个几何体的阴影被烘焙到贴图图像中。

定义　把阴影绘制到模型贴图上，将这种技术称为烘焙阴影。

因为这些图像是预先生成的(而不是在游戏运行时生成)，所以它们可以非常复杂、真实。缺点是，因为阴影提前生成，所以它们不能移动。因此，光照贴图非常适用于静态关卡几何体，而不适用于类似角色等动态对象。光照贴图自动生成而不用手工绘制。计算机计算场景中的灯光如何照亮关卡，而角落会出现微妙的暗影。在 Unity 中，渲染光照贴图的系统称为 Enlighten，可以在 Unity 的手册中查找该关键字。

使用实时阴影还是光照贴图，并不是一个非此即彼的选择。可以设置灯光的 Culling Mask 属性，使实时阴影只用于某些对象，允许在场景中将高质量的光照贴图用于其他对象。同样，虽然主角几乎总是投射阴影，但有时这个角色不应接收阴影，所有网格对象都有投射和接收阴影的设置(见图 8-6)。



图 8-6　Inspector 中投射和接收阴影的设置

定义　Culling 是移除不必要的东西的通用术语。这个词在计算机图形学的许多不同情况下使用。但在此，Culling Mask 是指想从阴影投射中移除的一系列对象。

现在明白了如何在场景中应用阴影的基础知识。关卡中的灯光和阴影本身是一个很大的主题(关于关卡编辑的书往往在光照贴图上就占用了很多章节)。这里只讨论打开一盏灯的实时阴影，之后讨论摄像机。

8.1.3　摄像机环绕玩家角色

在第一人称示例中，在 Hierarchy 视图中，摄像机和玩家关联在一起，所以它们会一起旋转。然而，在第三人称的移动示例中，玩家将独立于摄像机朝向不同的方向。因此这次不能在 Hierarchy 视图上把摄像机拖到玩家角色上。相反，摄像机的代码将跟随玩家角色移动其位置，但独立于角色做旋转。

首先，把摄像机相对于玩家放在希望的位置上，这里把位置设置为(0, 3.5, −3.75)，把摄像机放在玩家的后上方，如果有必要，把旋转重置为(0, 0, 0)。然后创建 OrbitCamera 脚本(参见代码清单 8.1)，将这个脚本组件添加到摄像机上，然后把玩家角色拖到这个脚本的 target 槽上。现在可以运行场景，看到摄像机代码的运行效果。

代码清单 8.1　观察目标时，绕着目标旋转的摄像机脚本

```
using UnityEngine;
using System.Collections;

public class OrbitCamera : MonoBehaviour {
    [SerializeField] private Transform target;      // 为环绕的对象序列化引用

    public float rotSpeed = 1.5f;

    private float _rotY;
    private Vector3 _offset;

    void Start() {
        _rotY = transform.eulerAngles.y;
        _offset = target.position - transform.position;      // 存储摄像机和目标之间的起始位置偏移
    }

    void LateUpdate() {
        float horInput = Input.GetAxis("Horizontal");      // 使用方向键缓慢旋转摄像机
        if (horInput != 0) {
            _rotY += horInput * rotSpeed;
        } else {
            _rotY += Input.GetAxis("Mouse X") * rotSpeed * 3;      // 或者使用鼠标快速旋转摄像机
        }

        Quaternion rotation = Quaternion.Euler(0, _rotY, 0);
        transform.position = target.position - (rotation * _offset);      // 维持起始偏移，根据摄像机的旋转进行位置偏移
        transform.LookAt(target);      // 不管摄像机在目标的什么地方，摄像机总是面向目标
    }
}
```

查看该代码清单时，要注意序列化的 target 变量。代码需要知道摄像机跟随哪个对象，所以序列化 target 变量，是为了让它出现在 Unity 编辑器上，然后把玩家角色和它关联起来。接下来的两个变量是旋转值，使用方式与第 2 章的摄像机控制代码的一样。代码还声明了一个_offset 值，它在 Start()函数中赋值，将摄像机存储到目标之间的偏移值。这样，在脚本运行时，就可以保持摄像机的相对位置。换句话说，无论哪个方向旋转，摄像机与角色之间将保持最初的距离。剩下的代码在 LateUpdate()方法中。

提示 *LateUpdate()方法是 MonoBehaviour 提供的另一个方法，类似于 Update()方法，每一帧都会运行它。这两个方法的区别在于，所有对象上运行了 Update()方法之后才执行 LateUpdate()方法，这样，就能确保在目标移动之后摄像机才更新。*

首先，代码基于输入控件来递增旋转值。代码查看两种不同的输入控件——水平方向键和水平鼠标移动——所以用一个条件来判断切换它们。代码检测是否按下水平方向键，如果是，那么使用这种输入，但如果不是，它会检查鼠标。通过分别检查这两种输入，代码可以为每一种输入类型以不同的速度旋转。

接下来，代码基于目标的位置和旋转值来定位摄像机。transform.position 那行代码可能是本段代码中最新奇的，因为它提供了前面未见过的重要数学知识。位置向量乘以四元数(quaternion)(注意，使用 Quaternion.Euler 方法将旋转角度转换为四元数)，结果是基于旋转的偏移位置。之后加上旋转后的位置向量，作为角色位置的偏移值来计算摄像机的位置。图 8-7 演示了计算的步骤，并详细分析了这行包含大量概念的代码。

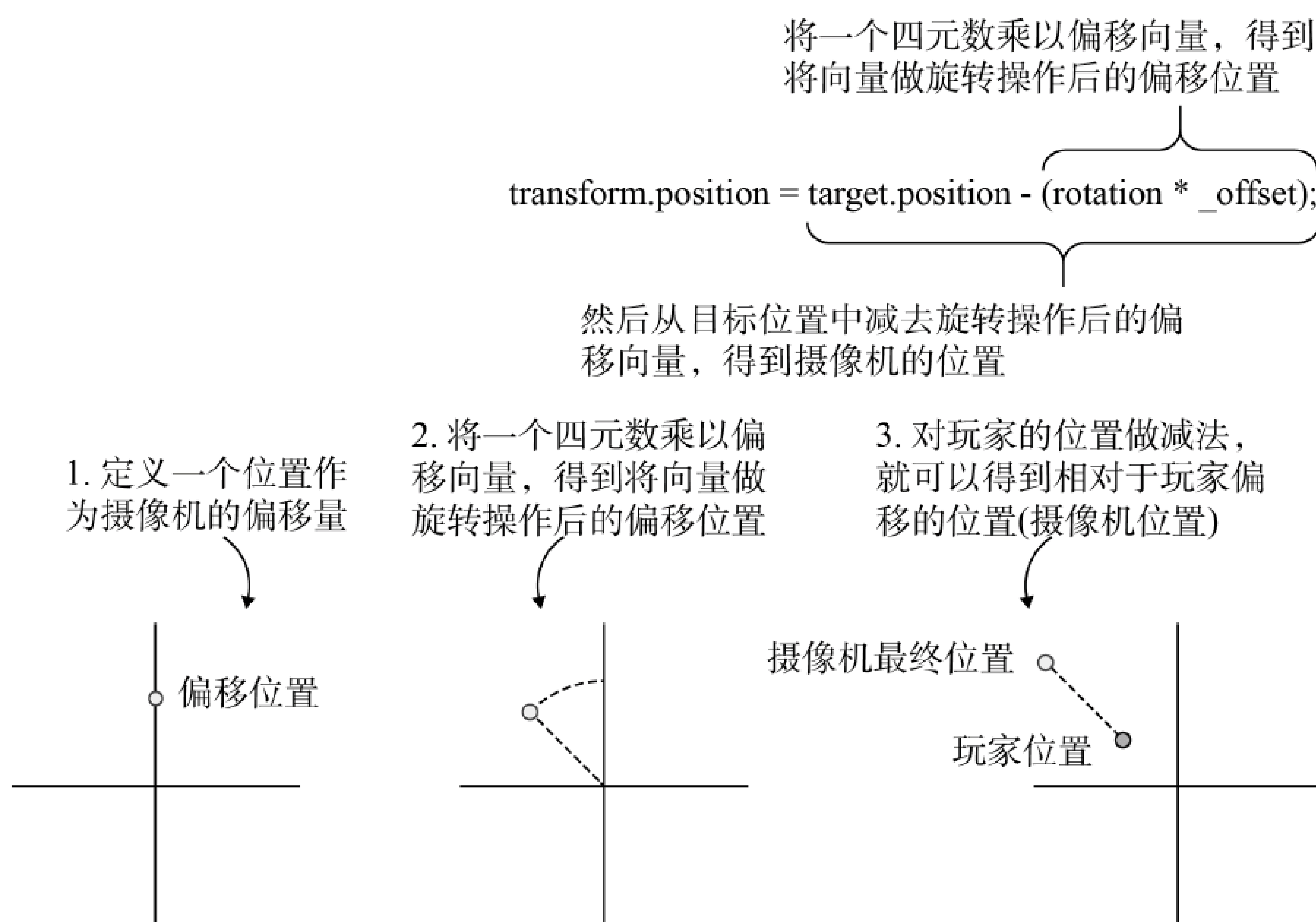


图 8-7　计算摄像机位置的步骤

注意 你可能会想到更精明的数学知识“嗯？第 2 章在坐标系统间变换的做法，在这里不能采用吗？”答案为“不能采用”。我们可以通过旋转坐标系来变换偏移位置，得到旋转的偏移量。但这需要预先设置好旋转的坐标系，而这一步是不必要的。

最后，代码中使用了 LookAt()方法使摄像机指向目标。这个方法使一个对象(不仅仅是摄像机)指向另一个对象。之前计算的旋转值用于定位摄像机，使它围绕目标位于正确的角度，但这一步中摄像机只是定位而没有旋转。因此，若没有调用 LookAt()方法这一步，摄像机只会环绕角色，但不会正好朝向它。注释掉“transform.position”那一行代码，看看会发生什么。

摄像机有实现环绕玩家角色的脚本，下一步是编写角色四处移动的代码。

8.2 编写程序控制摄像机的相对移动

现在角色模型已导入到 Unity，也编码实现了摄像机视图的控制，是时候编写代码来控制在场景中的四处移动了。下面编写代码控制摄像机的相对运动，当按下方向键时，让角色向不同方向移动。同时旋转角色，使它面向不同的方向。

“摄像机的相对运动”的含义

“摄像机的相对运动(camera-relative)”这个概念有点模糊不清，但理解它很重要。它类似于前面章节中提及的本地和全局的区别：在“对象的左边”和“整个世界的左边”中，“左”指向不同的方向。类似的，“移动角色到左边”，是指朝角色的左边移动，还是朝屏幕的左边移动？

第一人称游戏中的摄像机放在角色内，并跟着角色移动，所以摄像机的左边与角色的左边是没有区别的。第三人称视角把摄像机放在角色的外部，因此，摄像机的左边和角色的左边可能指向不同的方向。例如，如果摄像机朝向角色的前方，它们的左边就正好相反。因此，必须决定在特定的游戏和控件设置中想要什么。

大部分第三人称游戏都使控件相对于摄像机而运动，但偶尔会采用其他方式。当玩家按下左边的按钮时，角色移动到屏幕的左边，而不是角色的左边。随着时间的推移，游戏设计师尝试了不同的控制方案，发现当“左边”意味着“屏幕的左边”时，玩家觉得控制更直观，也更容易理解(这也是玩家的左边，并不是巧合)。

实现相对于摄像机的控制主要包括两个步骤：首先旋转玩家角色，以朝向控制的方向，然后向前移动角色。下面编写代码实现这两个步骤。

8.2.1 旋转角色，以朝向移动方向

首先，编写代码，让角色朝向方向键的方向。创建一个名为 RelativeMovement

的 C# 脚本(参见代码清单 8.2)。将该脚本拖到玩家角色上，然后把摄像机关联到脚本组件的 target 属性(就像把角色对象关联到摄像机脚本的目标一样)。现在按下方向键时，这个角色将相对于摄像机朝向不同的方向，而在没按任何方向键时站着不动(即用鼠标做旋转时)。

代码清单 8.2　相对于摄像机旋转角色

```
using UnityEngine;
using System.Collections;

public class RelativeMovement : MonoBehaviour {
    [SerializeField] private Transform target;

    void Update() {
        Vector3 movement = Vector3.zero;

        float horInput = Input.GetAxis("Horizontal");
        float vertInput = Input.GetAxis("Vertical");
        if (horInput != 0 || vertInput != 0) {
            movement.x = horInput;
            movement.z = vertInput;

            Quaternion tmp = target.rotation;
            target.eulerAngles = new Vector3(0, target.eulerAngles.y, 0);
            movement = target.TransformDirection(movement);
            target.rotation = tmp;

            transform.rotation = Quaternion.LookRotation(movement);
        }
    }
}
```

这个脚本需要引用相对移动的对象

从向量(0，0，0)开始并逐步添加移动组件

当按下方向键时只处理移动

保存初始旋转，以便在处理完目标对象后还原旋转

LookRotation() 计算了 movement 面向该方向的四元数

把 movement 的方向从本地坐标变换为世界坐标

代码清单 8.2 与代码清单 8.1 中使用了相同的方式，代码开头使用了一个序列化的 target 变量。就像之前的脚本需要一个相对它旋转的引用对象，这个脚本也需要一个可以相对它移动的引用对象。然后进入 Update()方法。该方法的第一行是声明了一个值为(0,0,0)的 Vector3。创建一个零向量，便于以后赋值，而不是以后用计算出来的移动值创建一个向量，这一点很重要。因为水平方向和垂直方向的移动值将在不同的步骤中计算，而它们都必须是同一向量的一部分。

和之前的脚本一样，下一步是检查输入控制。对于场景中的水平移动，需要设置移动向量的 X 和 Z 值。记得 Input.GetAxis()方法在没按下按钮时，返回 0；按下按钮时，它的值的变化范围是-1 和 1 之间。把 GetAxis()的返回值赋给移动向量，以确定沿着轴向的正方向还是负方向移动 (X 轴是左右方向，Z 轴是前后方向)。

接下来的几行代码将移动向量调整为相对于摄像机。具体而言，TransformDirection()方法用于将本地坐标转换为世界坐标。在第 2 章也使用了 TransformDirection()方法，但这次是变换目标的坐标系统，而不是玩家的坐标系统。同时，TransformDirection()这行前后的代码，根据需要对齐坐标系。首先保存目标的旋转，后面用于恢复，然后

调整旋转，使之仅绕 Y 轴旋转，而不是绕全部三根轴旋转。最后执行变换操作，并恢复目标的旋转。

所有代码都是用向量来计算移动方向的，最后一行代码通过 Quaternion.LookRotation() 方法将 Vector3 转换为 Quaternion，然后赋值给旋转，这样就能将移动方向应用到角色上。现在试着运行游戏，看看会发生什么！

使用插值平滑旋转

目前，角色的旋转会立即切换到不同的朝向，如果角色平滑旋转，会看起来好一些。为此可以使用一种称为插值的数学运算。首先给脚本添加一个变量:

```
public float rotSpeed = 15.0f;
```

把代码清单 8.2 中最后的那行 transform.rotation... 代码替换为如下代码:

```
      ...
      Quaternion direction = Quaternion.LookRotation(movement);
      transform.rotation = Quaternion.Lerp(transform.rotation,
         direction, rotSpeed * Time.deltaTime);
    }
  }
}
```

现在，不直接转型向LookRotation() 方法返回的值，而是把该值间接用作旋转的目标方向。使用Quaternion.Lerp()方法，在当前值和目标值之间平滑地旋转(第三个参数控制旋转的快慢)。

顺便说一下，值与值之间的平滑变化称为插值。可以在任何类型的两个值之间做插值，不仅仅是旋转值。Lerp 是“线性插值(linear interpolation)”的缩写，Unity 也提供了向量和浮点值的 Lerp 方法(插入位置、颜色或其他)。四元数也有一个密切相关的可替换的插值方法，称为Slerp(spherical linear interpolation，球形线性插值)。对于更慢的变化，Slerp 可能看起来比 Lerp 好一点。

当前，角色只是旋转而没有移动，下一节将为角色的移动添加代码。

注意　由于侧面朝向使用与环绕摄像机相同的键盘控制，因此当移动方向指向侧面时，角色会慢慢旋转。对控制的双重使用(doubling up)是这个项目所需的。

8.2.2　朝某方向移动

第 2 章为了让玩家在场景中四处移动，需要在玩家对象上添加一个角色控制器组件。为此选中角色，然后选择 Components | Physics | Character Controller。在 Inspector 中，应该把控制器的半径稍微减小到 0.4，除此之外，其他默认设置都适用于这个角色模型。

代码清单 8.3 显示了需要添加到 RelativeMovement 脚本中的内容。

代码清单 8.3　添加代码，改变玩家的位置

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))]
public class RelativeMovement : MonoBehaviour {
...
public float moveSpeed = 6.0f;

private CharacterController _charController;

void Start() {
    _charController = GetComponent<CharacterController>();
}

    void Update() {
        ...
          movement.x = horInput * moveSpeed;
          movement.z = vertInput * moveSpeed;
          movement = Vector3.ClampMagnitude(movement, moveSpeed);
          ...
        }

        movement *= Time.deltaTime;
        _charController.Move(movement);
    }
}
```

被方括号包围的这一行是放置 RequireComponent()方法的环境

该模式在前面的章节中介绍过，用于访问其他组件

覆盖已有的 X 和 Z 来应用移动速度

限制对角线移动的速度，使它和沿着轴移动的速度一样

总是将 movement 乘以 deltaTime，以使它们独立于帧率

如果现在运行这个游戏，会看到角色(保持 T 型姿势)在场景中移动。几乎整个代码清单都是之前看过的，所以在此简要回顾一下。

首先，在代码的开头有一个 RequireComponent()方法。如第 2 章所述，RequireComponent()方法会迫使 Unity 确保 GameObject 有一个传入命令的类型组件。这一行是可选的，不一定必须包含它，但如果没有，脚本会报错。

接着声明了移动值，随后获取了这个脚本的角色控制器的引用。在前面的章节中，GetComponent()会返回给定对象上依附的组件，如果进行搜索的对象没有显式定义，就假定是 this.gameObject.GetComponent()(即对象和脚本都一致)。

移动值是基于输入控制来分配的。之前的代码清单中也是如此。这里的区别在于，还考虑了移动速度。将移动轴乘以移动速度，并使用 Vector3.ClampMagnitude()将移动向量的大小限制为不超过移动速度。这个限制是必需的，否则对角线的移动就会比沿一个轴的移动更快(见直角三角形的斜边和两条边的图)。

最后，为了得到独立于帧率(frame rate-independent)的移动，将移动值乘以 deltaTime(“独立于帧率”的意思是，角色在帧速率不同的电脑上以相同的速度移动)。把移动值传给 CharacterController.Move()方法来实现移动。

处理了所有水平移动后，接下来介绍垂直移动。

8.3　实现跳跃动作

前一小节编写代码实现了角色在地面上四处移动。本章的引言也提过让角色跳跃，接下来实现这个功能。大多数第三人称游戏都有一个跳跃的控制。即使没有，也几乎总是有角色从悬崖上降落时做垂直移动。我们的代码将处理跳跃和降落。具体而言，代码总是利用重力把玩家往下拉，偶尔当玩家跳跃时，应用一个向上的跳动。

在编写代码之前，先给场景添加一些升起的平台。目前没有可以跳上去或者掉落下去的平台。创建一对立方体对象，然后修改它们的位置和比例，作为玩家跳跃的平台。在示例项目中，添加了两个立方体，使用了如下设置：位置(5, 0.75, 5)和比例(4, 1.5, 4)；位置(1, 1.5, 5.5)和比例(4, 3, 4)。图 8-8 显示了升起的平台。

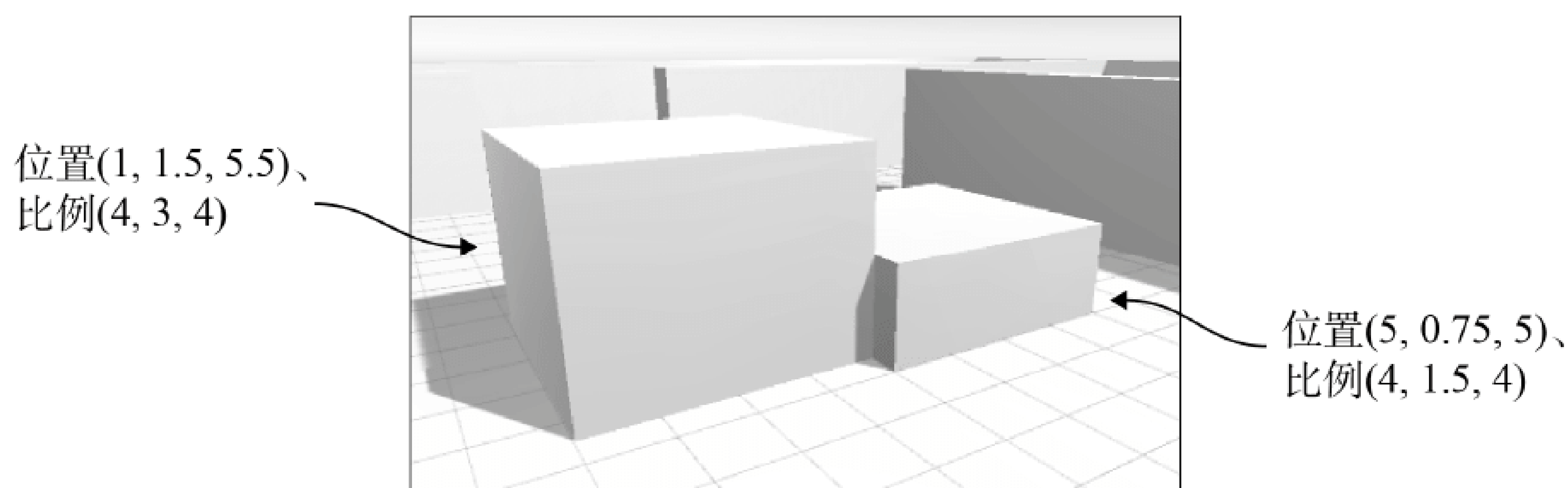


图 8-8　将两个升起的平台添加到场景

8.3.1　应用垂直速度和加速度

如前所述，第一次开始写代码清单 8.2 中的 RelativeMovement 脚本时，分步骤计算了移动值，并逐步将这些值添加给移动向量。代码清单 8.4 将垂直移动添加给现有的向量。

代码清单 8.4　将垂直移动添加给 RelativeMovement 脚本

```
...
public float jumpSpeed = 15.0f;
public float gravity = -9.8f;
public float terminalVelocity = -10.0f;
public float minFall = -1.5f;

private float _vertSpeed;
...
void Start() {
    _vertSpeed = minFall;          ← 在已有的 Start()方法中将垂直速度初始化为最小下落速度
    ...
}
    void Update() {
        ...
        if (_charController.isGrounded) {     ← CharacterController 的 isGrounded 的属性用于检查控制器是否在地面上
            if (Input.GetButtonDown("Jump")) {     ← 当在地面时响应 Jump 按钮
                _vertSpeed = jumpSpeed;
```

```
            } else {
                _vertSpeed = minFall;
            }
        } else {
            _vertSpeed += gravity * 5 * Time.deltaTime;
            if (_vertSpeed < terminalVelocity) {
                _vertSpeed = terminalVelocity;
            }
        }
        movement.y = _vertSpeed;

        movement *= Time.deltaTime;
        _charController.Move(movement);
    }
}
```

如果不在地面上，那么应用重力，直到垂直速度达到了终止速度

可以在代码清单 8.3 的结尾看到这行代码

像往常一样，开始时在脚本的顶部为各种移动值添加一些新变量，并正确初始化这些值。然后直接跳到一个用于水平移动的长if 语句之后，在这里添加另一个用于垂直移动的长 if 语句。具体而言，代码将检测角色是否在地面上，因为垂直速度将根据角色是否在地面上做不同的调整。CharacterController 中的 isGrounded 属性用于检测角色是否在地面上。如果角色控制器的碰撞器在最后一帧与任何东西碰撞，这个值就为 true。

如果角色在地面上，那么垂直速度的值(私有变量_vertSpeed)基本上重置为 0。角色在地面上时不会下落，所以显然它的垂直速度是 0。如果角色走下悬崖，角色就会自由下落，因为它降落的速度将加快。

注意 角色在地面上时的垂直速度不完全为 0，实际上，垂直速度设置为 minFall，这是一个轻微向下的移动速度，这样角色水平移动时总是被按在地面上。角色需要一个向下的力才能在凹凸不平的地面上行走。

垂直速度的另一种情况是单击了跳跃按钮，在这种情况下，垂直速度应该设置为一个较高的数字。if 条件语句检查 GetButtonDown()函数，这个输入函数的作用与 GetAxis()一样，返回指定的控制输入的状态。与 Horizontal、Vertical 输入轴一样，真正分配给 Jump 的键在 Edit | Project Settings 的 Input 设置中定义(默认分配的是 Space——空格键)

回到上面的长 if 条件语句，如果角色没有在地面上，那么垂直速度应该不断地因重力而减小。注意这段代码不是单纯设置速度值，而是递减这个值。这样它就是一个恒定的速度，而是有一个向下的加速度，得到真实的下降移动。随着角色的上升速度逐渐降低为 0，角色开始下降，跳跃就出现一条自然的弧线。

最后，代码确保向下的速度不超过最终速度。注意，操作符是“小于”而不是“大于”，因为向下速度为负值。接着在长 if 条件语句之后，把计算好的垂直速度赋给移动向量的 Y 轴。

这就是真实的垂直移动。当角色不在地面上时，应用一个向下加速度的常量。当角色在地面上时，适当地调整速度。这段代码实现了很不错的下降行为。但这一切都取决于正确地检测地面，并且需要修复一个小故障。

8.3.2　修改地面检测来处理边缘和斜坡

如上一节所述，CharacterController 的 isGrounded 属性表明在最后一帧 4 角色控制器的底部是否和任何东西碰撞。虽然这种检测地面的方法在大部分时间是有效的，但注意，角色走到边缘上时似乎漂浮在空中，这是因为角色的碰撞区域是个胶囊(当选中角色对象时，可以看到这种效果)。当玩家走下平台的边缘时，胶囊的底部仍然与地面接触。图 8-9 说明了这个问题。这并不是我们想要的效果!

同样，如果角色站在斜坡上，当前的地面检测将导致有问题的行为。现在尝试创建一个倾斜块靠着高出的平台。创建一个新的立方体对象，然后设置它的变换值：Position 为(−1.5, 1.5, 5)，Rotation 为(0, 0, −25)，Scale 为(1, 4, 4)。

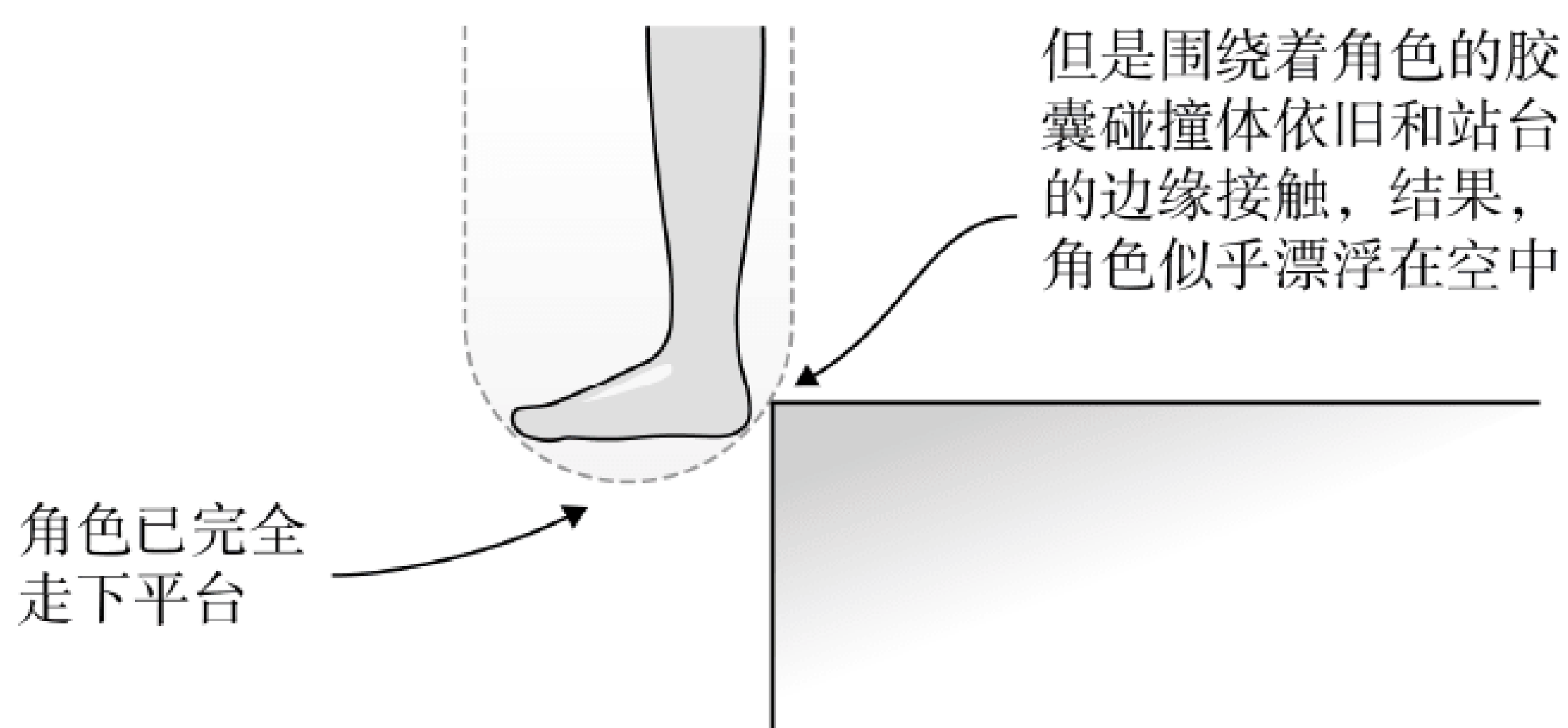


图 8-9　角色控制器胶囊与平台边缘的碰撞

如果从地面跳上斜坡，则将发现要从斜坡的中途多跳一次，才能登上顶部。这是因为斜坡触碰到了胶囊的底部，然而当前的代码把底部的任何碰撞都当成是角色已经站稳了。同样，这也不是我们想要的效果。这个角色应该滑下来，因为没有可以起跳的坚实立足点。

注意　只需要从陡峭的斜坡上滑下来，而在比较平缓的斜坡上，如凹凸不平的地面等，玩家的行走应不受影响。如果想要进行一个测试，创建一个立方体，则制作一个平缓的斜坡，然后设置 Position 为(5.25, 0.25, 0.25)，Rotation 为(0, 90, 75)，Scale 为(1, 6, 3)。

导致所有这些问题的根本原因都相同：检测角色底部的碰撞体来决定角色是否在地面上，这并不是一种很好的方式。这里使用光线投射来检测地面。第 3 章的 AI 使用光线投射来检测前方的障碍物，下面用同样的方法来检测角色下方的表面。在角色的位置下方投射光线，如果它在角色的脚下产生碰撞，就意味着玩家站在地上。

这引入了一个需要处理的新情况：光线投射没有检测角色下方的地面，但角色控制器碰撞到了地面。如图 8-9 所示，当角色走下边缘时，胶囊仍与平台碰撞。图 8-10 增加了光线投射，以便展示现在所发生的情形：光线没有击中平台，但是胶囊确实触碰到边缘。代码需要处理这一特殊情况。

图 8-10 当走下边缘时光线往下投射

在这种情况下，代码应该让角色从边缘滑落，角色仍然会降落(因为它没有站在地面上)，但它也会从碰撞点推离(因为它需要从与之碰撞的平台上移开胶囊)。那么，代码将用角色控制器检测碰撞，并通过将角色推离碰撞点来响应碰撞。

代码清单 8.5 结合了刚刚讨论的内容，修改了垂直移动。

代码清单 8.5 使用光线投射检测地面

```
...
private ControllerColliderHit _contact;        ← 需要在函数之间存储碰撞数据
...
        bool hitGround = false;
        RaycastHit hit;
        if (_vertSpeed < 0 &&        ← 检查玩家是否在掉落
            Physics.Raycast(transform.position, Vector3.down, out hit)) {
            float check =        ← 检查碰撞的距离，稍微超过胶囊体的底部
                (_charController.height + _charController.radius) / 1.9f;
            hitGround = hit.distance <= check;
        }
                                      检查光线投射结果，
        if (hitGround) {        ← 代替 isGrounded 检查
            if (Input.GetButtonDown("Jump")) {
                _vertSpeed = jumpSpeed;
            } else {
                _vertSpeed = minFall;
            }
        } else {
            _vertSpeed += gravity * 5 * Time.deltaTime;
            if (_vertSpeed < terminalVelocity) {
                _vertSpeed = terminalVelocity;
            }
                                               光线投射没有检测到地面，
            if (_charController.isGrounded) {        ← 但胶囊体接触到了地面
```

```
                if (Vector3.Dot(movement, _contact.normal) < 0) {
                    movement = _contact.normal * moveSpeed;
                } else {
                    movement += _contact.normal * moveSpeed;
                }
            }
        }
        movement.y = _vertSpeed;

        movement *= Time.deltaTime;
        _charController.Move(movement);
    }

    void OnControllerColliderHit(ControllerColliderHit hit) {
        _contact = hit;
    }
}
```

根据角色是否面向接触点，响应稍微不同

当检测碰撞时将碰撞数据保存在回调中

代码清单 8.5 包含了代码清单 8.4 中很多相同的代码，新代码被穿插到现有的移动脚本，此代码清单需要现有代码作为上下文。第一行将一个新变量添加到 RelativeMovement 脚本的顶部。这个变量用于在函数之间保存碰撞数据。

接下来的几行代码做光线投射。这段代码在水平移动之下、垂直移动的 if 语句之上。Physics.Raycast()的实际调用在前面的章节中介绍过，但在此具体的参数有所不同。虽然投射光线的位置相同(即角色的位置)，但这次的方向是向下而不是向前。之后，当光线碰撞到某物时检查光线投射的距离，如果碰撞距离在角色的脚附近，那么角色就站在地面上，因此将 hitGround 设置为 true。

警告　如何计算检查的距离有点不明显，所以需要仔细地分析一下。首先将角色控制器的高度(这个高度不包含球面端)加上球面端，把这个值除以 2，因为光线从角色的中心投射(也就是说，已经是一半了)，获取到角色底部的距离。但真正要检查的距离则要超出角色底部一点点，考虑到光线投射微小的不准确性，因此除以 1.9 而不是除以 2，得到稍微远点的距离。

做完光线投射后，在垂直移动的 if 语句中使用 hitGround 代替 isGrounded。大部分垂直运动的代码将保持不变，但是要添加代码，处理玩家不在地面上(即玩家走下平台的边缘)时，角色控制器触碰地面的情况。这里添加了 isGrounded 条件，但注意该条件嵌套在 hitGround 条件中，目的是 isGrounded 只检查 hitGround 没有检测到地面的情况。

碰撞数据包含了一个 normal 属性(法向量表示物体的朝向)，该属性指定了推离碰撞点的方向。但一个棘手的问题是，我们想以不同的方式从碰撞点推离，这取决于玩家正在移动的方向。当先前的水平移动朝向平台时，就要替换掉该移动，目的是角色不会一直朝着错误的方向移动。但当角色背向边缘时，就应添加到先前的水平移动量，以继续前进，远离边缘。移动向量相对碰撞点的朝向可以使用点积来决定。

定义　点积是一种作用在两个向量上的数学运算，简言之，两个向量的点积在-1 到 1 之间。1 意味着它们指向相同的方向，-1 则意味着它们指向相反的方向。不要混淆点积和叉积，叉积是一个不同的、常见的向量数学运算。

Vector3 包含了 Dot()函数，用于计算两个给定向量的点积。如果计算移动向量和碰撞点的法向量之间的点积，当两个方向相反时，则返回一个负数，而当移动和碰撞点的方向相同时，则返回正数。

最后，在代码清单 8.5 的末尾给脚本添加了一个新方法。在前面的代码中检查了碰撞的法向量，但这些信息从何而来？角色控制器的碰撞信息通过 MonoBehaviour 提供的 OnControllerColliderHit()回调函数来报告，为了响应脚本其他地方的碰撞数据，该数据必须保存在外部变量中。该函数的作用在于：将碰撞数据保存在_contact 中，以便该数据可以在 Update()方法中使用。

现在，平台边缘和斜坡上的错误已修正完毕。下面继续运行游戏并测试，跨过边缘和跳上陡峭的斜坡。这个移动的例子几乎已完成。角色在场景中已能正确移动，所以只剩下一件事：给角色添加动画，去掉 T 型姿势。

8.4　设置玩家角色上的动画

除了由网格几何体定义的更复杂的形状外，还需要给人物角色设置动画。第 4 章提到，动画是一个信息包，它定义了相关的 3D 对象的运动。当时给出的例子是一个角色在行走，而这正是目前要处理的情况。角色在场景中行走，所以要给它分配动画，让它的手臂和腿来回摇摆。图 8-11 显示了给角色设置动画后它在场景中移动时的情形。

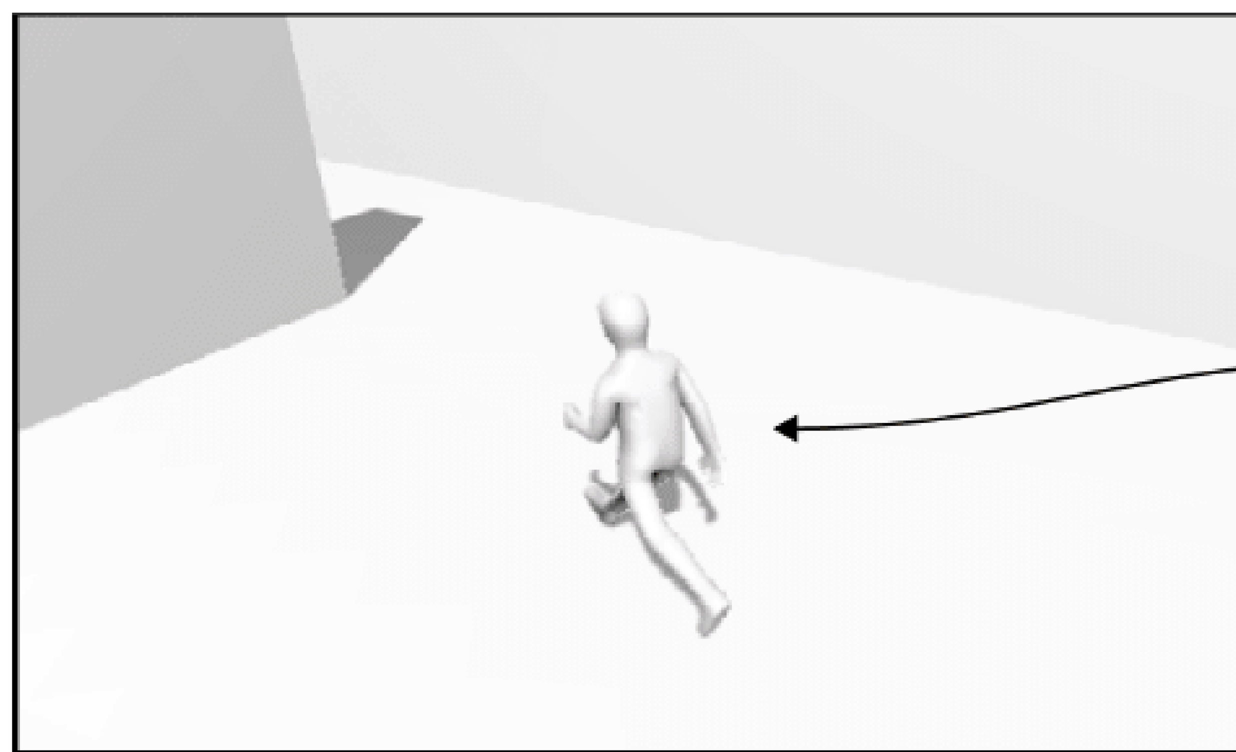


图 8-11　角色随着动画的播放而移动

理解 3D 动画的一个很好的比喻是思考操纵木偶的人：3D 模型是木偶，动画机是操纵木偶的人，而动画是木偶运动的一些记录。动画可以通过几种不同的方法来创建，现在游戏的大部分角色动画(当然包括本章的所有角色动画)都使用一种称为骨骼动画(skeletal animation)的技术。

定义　骨骼动画是一种动画，在模型中一系列骨骼建立，然后在动画期间骨骼在四周运动。当某块骨骼运动时，模型上与该骨骼关联的表层也跟着一起运动。

骨骼动画通过模拟角色内部的骨骼来产生最直观的效果(图 8-12 阐释了这一点)，但骨骼是一种抽象，只要希望模型在弯曲的同时仍以确定的结构移动(例如摆来摆去的触手)就可以使用它。虽然骨骼移动很生硬，但骨骼外部的模型表层可以弯曲。

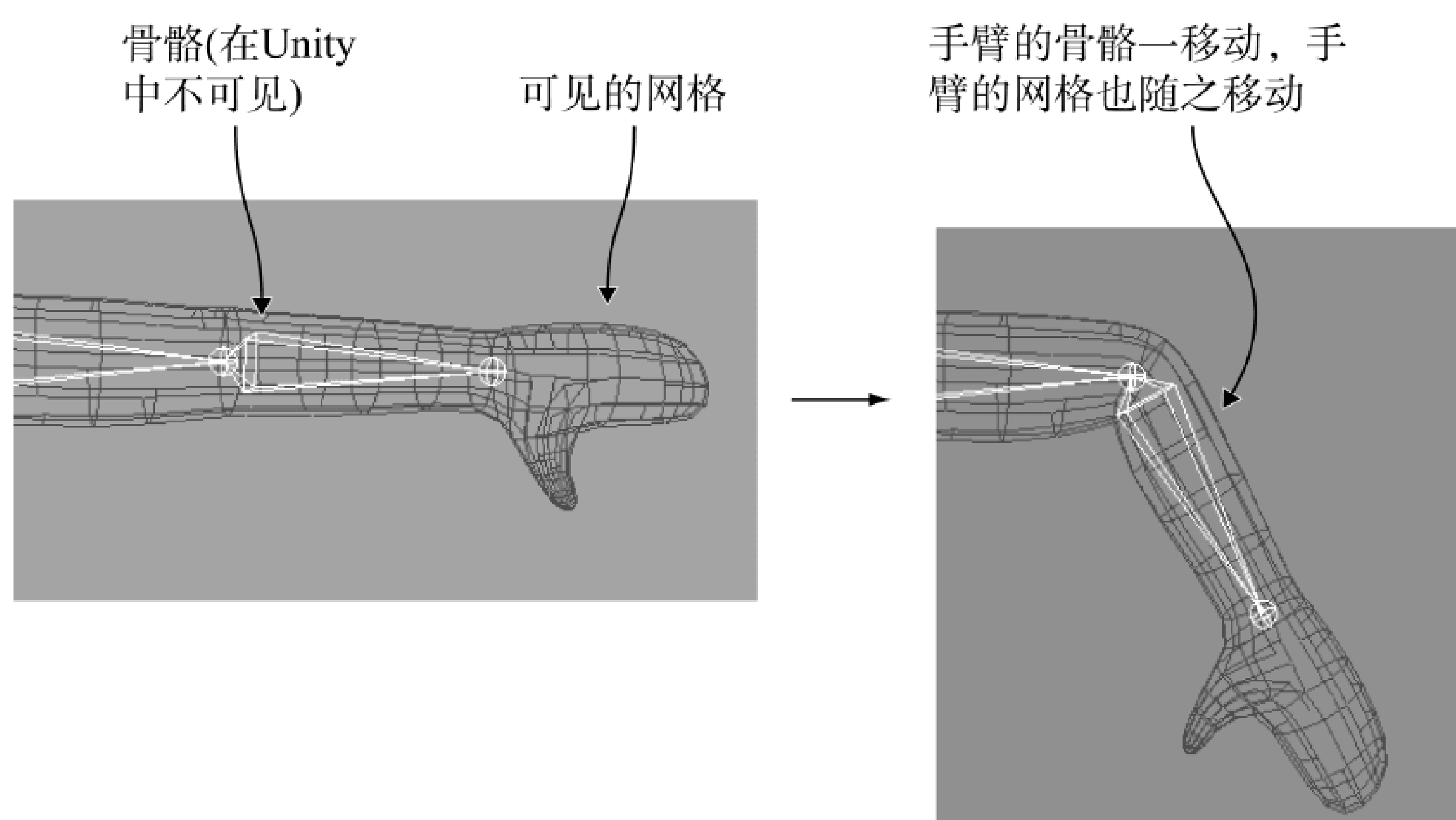


图 8-12　人物角色的骨骼动画

实现如图 8-11 所示的结果包括以下几个步骤：首先在导入的文件中定义动画剪辑(animation clips)，然后设置控制器来播放这些动画剪辑，最后把动画控制器并入到代码中。角色模型上的动画将会按照移动脚本进行回放。

当然，在完成这些步骤之前，首先要打开动画系统。在 Project 视图中选择玩家模型，在 Inspector 中查看它的 Import 设置，选择 Animations 选项卡，确保 Import Animation 复选框被选中。之后选择 Rig 选项卡，将 Animation Type 从 Generic 切换到 Humanoid(自然，这是一个人物角色)。注意，这个菜单中也有一个 Legacy 设置，Generic 和 Humanoid 都是在 Mecanim 动画系统范畴的设置。

解释 Unity 的 Mecanim 动画系统

Unity 有一个管理模型动画的复杂系统，称为 Mecanim。第 6 章介绍了这个动画系统，并指出后面将进行更详细的介绍，因此本章将回顾以前的解释，现在将重点放在 3D 动画而不是 2D 上。Mecanim 这个特殊的名字标识它是一个更新、更高级的动画系统，添加到 Unity 中用于替代旧的动画系统。旧的系统依旧存在，被标识为 Legacy 动画，但它可能在未来的 Unity 版本逐步被淘汰。那时，Mecanim 是 Unity 中唯一的动画系统。

虽然要使用的动画都包含在和角色模型一样的 FBX 文件中，但 Mecanim 最主要的好处之一是将其他 FBX 文件中的动画应用到角色上。例如，所有人形敌人都可以共享一组动画。这有许多优点，包括所有的数据很有组织(模型可以放在一个文件夹中，而动画放在另一个文件夹中)，同时能节约为每个角色独立制作动画的时间。

单击 Inspector 面板底部的 Apply 按钮，锁定导入模型的设置。然后继续定义动画剪辑。

警告　注意控制台中的一个警告“conversion warning: spine3 is between humanoid transforms.”，不必担心这个警告，它表明 Mecanim 预测在导入的模型上有额外的骨骼。

8.4.1　在导入的模型上定义动画剪辑

设置角色动画的第一步是定义各种可播放的动画剪辑。如果考虑一个栩栩如生的角色在不同的时间会出现不同的运动：有时玩家跑来跑去，有时玩家跳上平台，有时角色只是站立在那，手臂往下放。这些运动都是一个可以单独播放的独立“剪辑”。

通常，导入的动画是单个很长的剪辑，它可以剪切成多个短小的单独动画。为了分离动画剪辑，首先选择 Inspector 上的 Animations 选项卡，这会显示 Clips 面板，如图 8-13 所示，其中列出了所有已定义好的动画剪辑，而这最初就是一个导入的剪辑。注意底部的+和−按钮，可以使用这两个按钮来添加或删除列表中的剪辑。最终需要为这个角色制作 4 个剪辑，所以根据需要来添加或删除剪辑。

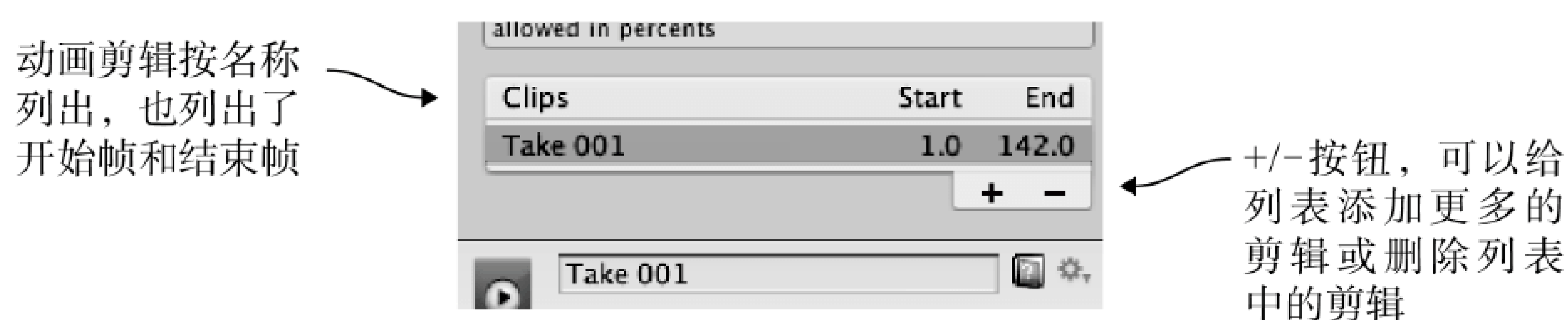


图 8-13　Animation 设置中的剪辑列表

选择一个剪辑后，该剪辑的相关信息(如图 8-14 所示)就会出现在列表的下方。剪辑的名称显示在剪辑信息区域的顶部，可以输入新的剪辑名称。把第一个剪辑命名为 idle，为这个动画剪辑定义开始帧和结束帧，这允许从导入的长动画中切出一小段动画。将 idle 动画的 Start 设置为 3，End 设置为 141。下一步介绍 Loop 设置。

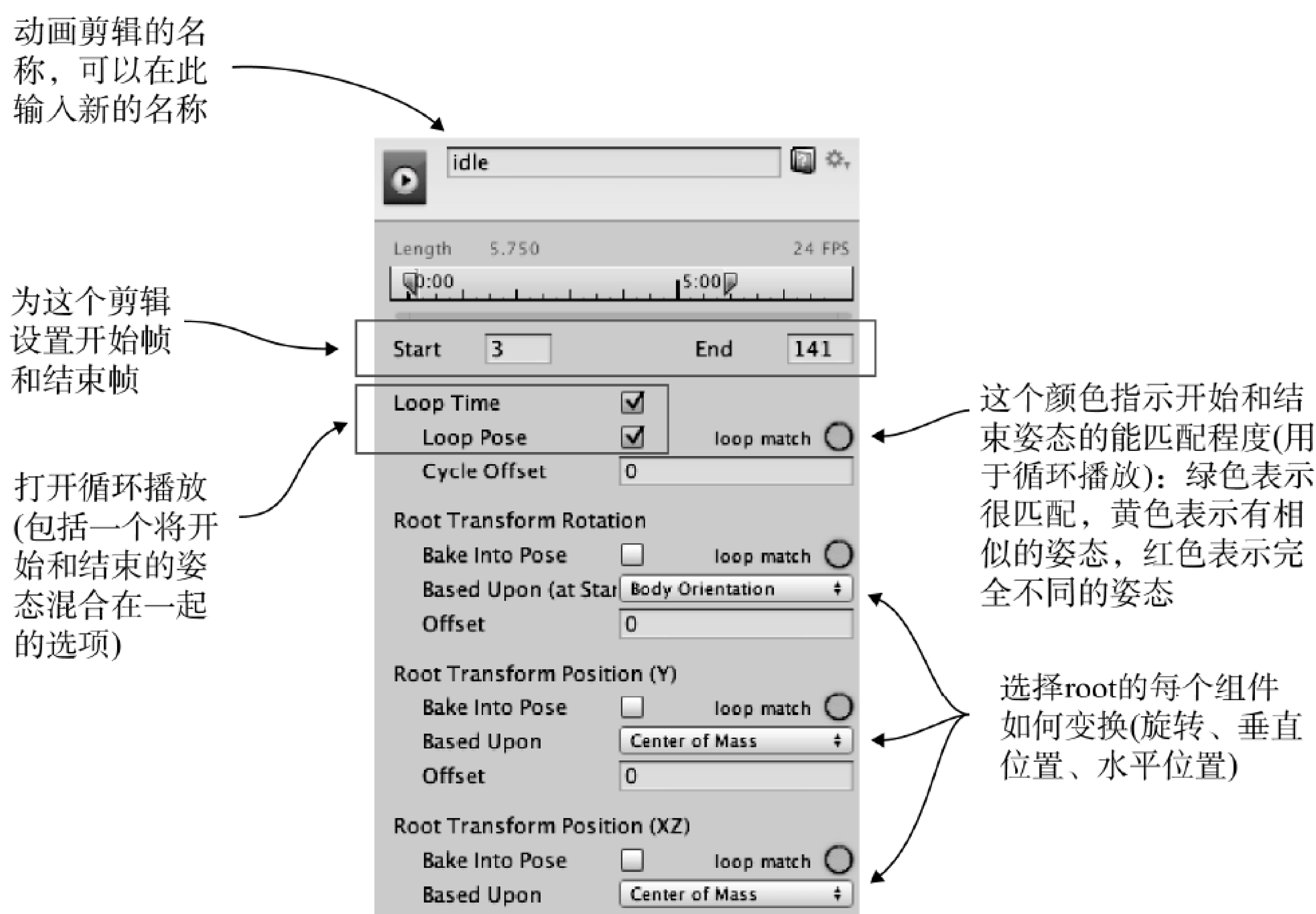


图 8-14　被选中的动画剪辑的信息

定义 Loop 指的是一条反复多次播放的记录。循环的动画剪辑播放结束后，再次从开头播放。

idle 动画循环时，需要同时选择 Loop Time 和 Loop Pose。顺便说一下，绿色指示器表明剪辑的开始帧和最后帧的姿态匹配，可以正确循环。当姿态有点出入时，指示器变为黄色。当开始帧和结束帧完全不匹配时，指示器变为红色。

在 Loop 设置的下方有一些与 root 变换相关的设置。root 这个词意味着对骨骼动画所做的操作与 Unity 中的级联对象相同：root 对象是连接一切其他对象的基础对象。因此动画的 root 是角色的基点，一切其他对象都相对于它移动。这里有关于基点的一些不同设置，可以对动画实验这些设置。对本例而言，三个菜单的设置应该按如下顺序进行：Body Orientation、Center Of Mass、Center Of Mass。

现在单击 Apply，给角色添加一个 idle 动画剪辑。用相同的操作再做两个剪辑：walk 剪辑开始帧 144，结束帧 169；run 剪辑开始帧 171，结束帧 190。因为它们都是循环动画，所以其他设置与 idle 剪辑的一样。

第四个动画剪辑是跳，它的设置有点不同。首先，它不是一个循环而是单个姿态，所以不选择 Loop Time。设置开始帧到结束帧为 190.5 到 191。虽然它是单帧姿态，但 Unity 要求开始帧和结束帧是不同的。因为这些数据比较棘手，所以下方的动画预览显示不正确，但游戏中的姿态看起来还不错。

单击 Apply，确认新的动画剪辑已完成，然后跳到下一步：创建动画控制器。

8.4.2　为动画创建动画控制器

这一步是为角色创建动画控制器，允许创建动画状态，创建状态之间的变换。不同动态状态下播放不同的动画剪辑，然后使用脚本来控制动画状态之间的变换。

这个间接关系有点奇怪——在代码和实际播放动画之间放上控制器的抽象。我们熟悉直接在代码中播放动画的系统，实际上，Legacy 动画系统完全采用这种方式，如 Play("idle")。但这种间接关系允许模型共用一个动画，而不是只能在模型内部播放动画。本章不利用这种功能，但要记住，处理大型项目时，这种功能很有优势。可以从多个来源得到动画，包括多个动画控制器，也可以在 Unity 的在线商店购买单独的动画（如 Unity 的 Asset Store）。

首先创建一个新的动画控制器资源(Assets | Create | Animator Controller——不是 Animation，它们是不同类型的资源)。在 Project 视图上会看到一个图标，其中包含了一个看起来很有趣的网格线，把这个资源重命名为 player，如图 8-15 所示。选择场景中的角色，注意这个角色上有一个名为 Animator 的组件。可以制作动画的任何模型都有这个 Animator 组件，当然还有 Transform 组件和用户添加的其他组件。这个 Animator 组件包含一个可以关联特定动画控制器的 Controller 槽，所以可以将新的动画控制器

资源拖放到该槽中(请确保不要选中 Root Motion)。



图 8-15　动画控制器和 Animator 组件

动画控制器是一棵连接节点的树(因此该资源的图标也表示这个含义)，打开 Animator 视图可以查看和操作它。Animator 视图类似于 Scene 或 Project 的视图(如图 8-16 所示)，但该视图默认是不打开的。在 Window 菜单中选择 Animator(注意，不要与 Animation 窗口相混淆，Animation 是一个独立于 Animator 的选项)可以打开它。在此显示的这个节点网络是指当前选择的动画控制器(或者所选角色上的动画控制器)。

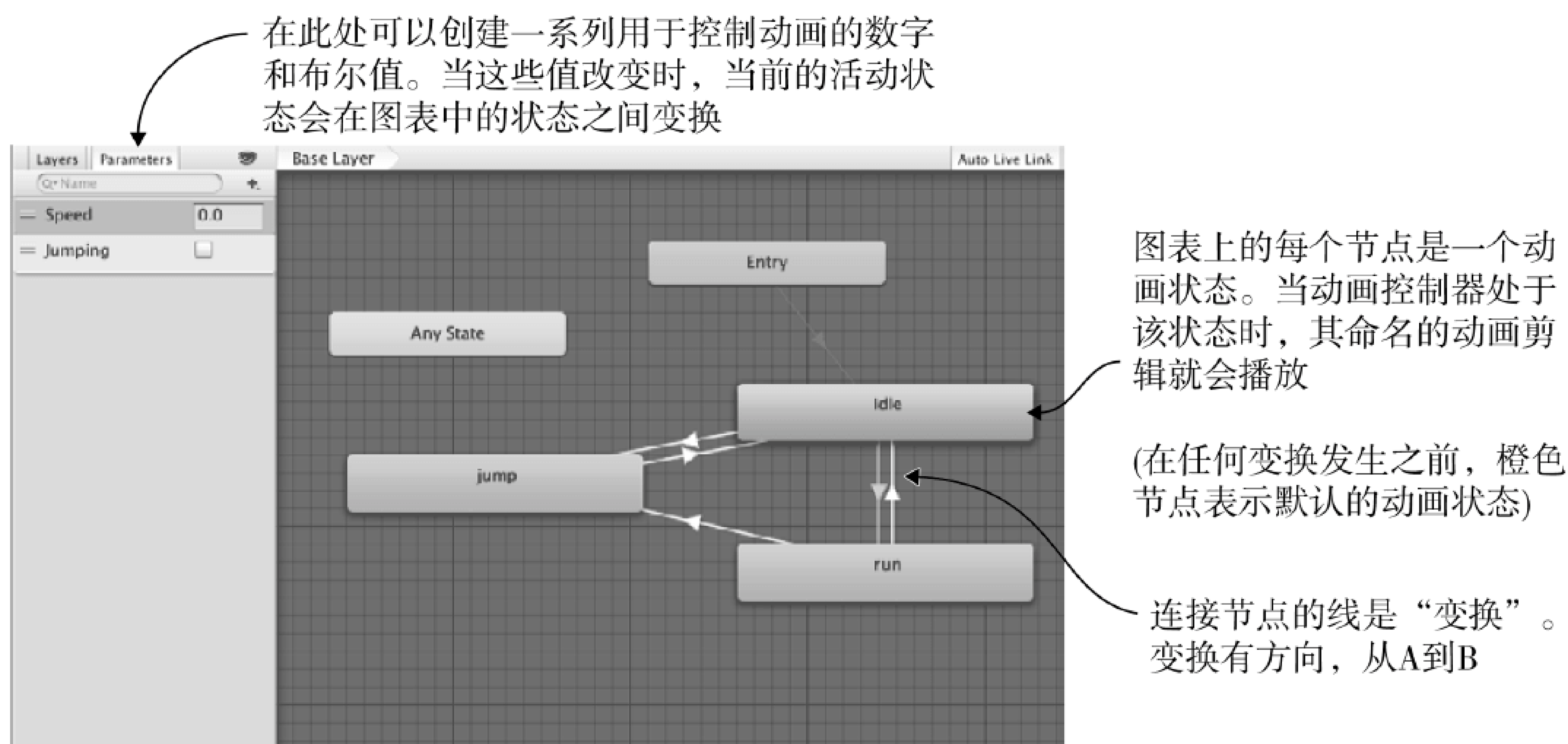


图 8-16　Animator 视图中已完成的动画控制器

提示　可以在 Unity 中移动选项卡，将它们停靠在需要的位置来组织界面。作者喜欢将 Animator 选项卡停靠在 Scene 和 Game 窗口旁边。

起初只有两个默认节点 Entry 和 Any State。本例不使用 Any State 节点，而是拖入动画剪辑来创建新的节点。在 Project 视图中，单击模型资源边上的箭头，展开看看它包含了什么内容。这个资源的内容是已定义的动画剪辑(如图 8-17 所示)，把这些动画剪辑拖入 Animator 视图中。除了 walk 剪辑外，可以拖入 idle、run 和 jump 剪辑(walk 剪辑可以用于其他项目)。



图 8-17　在 Project 视图中展开模型资源

右击 idle 节点，选择 Set As Layer Default State。该节点会变成橙色，而其他节点保持灰色。在游戏做出任何改变之前，默认的动画状态就是这个节点网络的起点。需要用线条将节点连在一起，指示动画状态之间的变换。右击节点，选择 Make Transition 以拖出一个箭头，然后单击另一个节点，就将这两个节点连接好。这样的连接节点如图 8-16 所示 (确保大部分节点在两个方向上变换，但从 jump 到 run 不是两个方向上变换)。这些变换线条决定了动画状态如何相互连接，控制在游戏中从一个状态到另一个状态的变换。

变换依靠一组控制值，下面创建这些参数。在图 8-16 的左上角有一个 Parameters 选项卡。单击它，可以看到面板上有添加参数的+按钮。添加一个浮点值 Speed 和一个布尔值 Jumping。可以通过代码调整这些值，它们会触发动画状态之间的变换。

单击变换线，在 Inspector 中查看它们的设置(见图 8-18)。在此可以调整当参数值改变时动画状态的变化方式。例如，单击 idle-to-run 上的变换线可以修改变换的条件。在 Conditions 下，选择 Speed、Greater 和 0.1。关闭 Has Exit Time(它会迫使动画一直播放，而不是在变换发生时马上暂停)。然后单击 Settings 选项卡旁边的箭头，以便看到整个菜单，其他状态应该可以打断当前变换状态，因此把 Interruption Source 菜单从 None 改为 Current State。对于表 8-1 的所有动画变换，重复这一步。

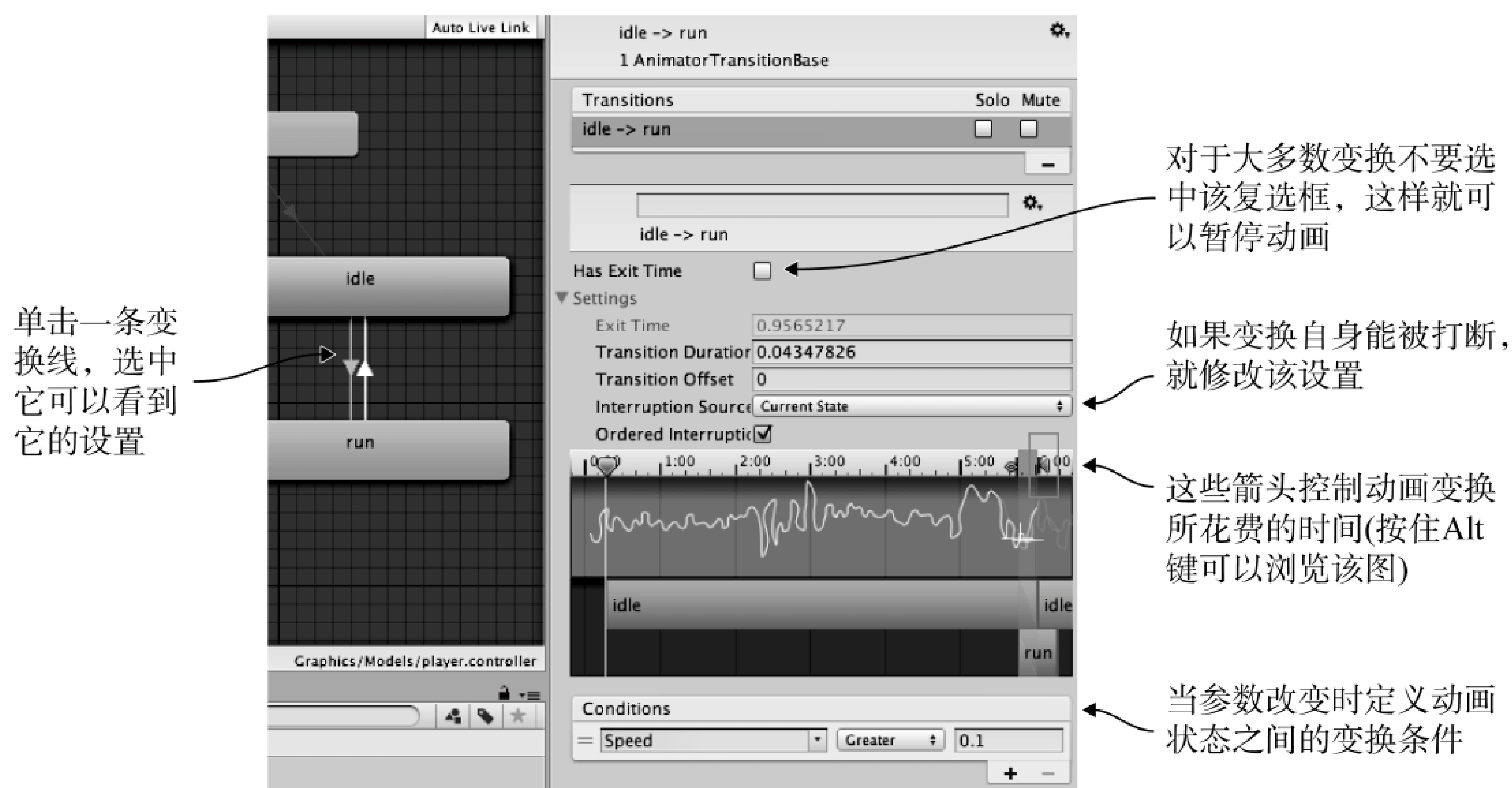


图 8-18　在 Inspector 中的 Transition 设置

表 8-1　在动画控制器中的所有变换条件

变　换	条　件	打　断
idle-to-run	速度大于 0.1	当前状态
run-to-idle	速度小于 0.1	无
idle-to-jump	Jumping 为 true	无
run-to-jump	Jumping 为 true	无
jump-to-idle	Jumping 为 false	无

除了这些基于菜单的设置，在 Conditions 设置的上方还有一个复杂的可视化界面，如图 8-18 所示。该图允许可视化地修改变换的时间长短。对于 idle 和 run 之间的变换，默认的动画变换时间看起来很合理，但在往返于 jump 的所有变换应该更短些，以便角色能更快地切换到 Jump 动画，以及从 Jump 动画更快地切换出来。图 8-18 的阴影区域表示动画变换需要多长时间。为了看到更多的细节，可以使用 Alt+鼠标左键单击平移图表，Alt+鼠标右键单击缩放图表(这些操作与在 Scene 视图中的导航类似)。对 3 个 jump 的变换，在阴影区域的顶部上面用鼠标将阴影区缩小到 4 毫秒。

最后，一次选择一个动画节点，调整变换的排序，来完善这个动画网络。在 Inspector 上会显示往返于该节点的所有变换，可以拖动这个列表中的项(它们的拖动手柄是左边的图标)重新排序。确保 jump 的动画变换在 idle 和 run 节点的上方，这样 jump 的变换就优先于其他变换。查看这些设置，如果觉得动画的播放有点慢，可以修改播放速度(运行速度 1.5 看起来更好)。

自此，动画控制器已设置完毕，现在可以通过移动脚本来操作这些动画。

8.4.3　编写操作 Animator 组件的代码

最后，给 RelativeMovement 脚本添加一些方法。如前所述，设置动画状态的大部分工作已在动画控制器中完成，现在只需要少量代码来操作丰富多变的动画系统(见代码清单 8.6)。

代码清单 8.6　在 Animator 组件中设置值的代码

```
...
private Animator _animator;
...
_animator = GetComponent<Animator>();      ← 在 Start()函数中添加
...
        _animator.SetFloat("Speed", movement.sqrMagnitude);   ← 正好位于水平移动的 if 语句下

        if (hitGround) {
            if (Input.GetButtonDown("Jump")) {
                _vertSpeed = jumpSpeed;
```

```
            } else {
               _vertSpeed = -0.1f;
                _animator.SetBool("Jumping", false);
            }
        } else {
            _vertSpeed += gravity * 5 * Time.deltaTime;
            if (_vertSpeed < terminalVelocity) {
                _vertSpeed = terminalVelocity;
            }
            if (_contact != null ) {                          ← 不要在关卡的开始
                _animator.SetBool("Jumping", true);              处触发这个值
            }

            if (_charController.isGrounded) {
                if (Vector3.Dot(movement, _contact.normal) < 0) {
                    movement = _contact.normal * moveSpeed;
                } else {
                    movement += _contact.normal * moveSpeed;
                }
            }
        }
...
```

同样，代码清单 8.6 与前面的代码清单有很多重复之处，动画代码是小部分穿插在现有移动脚本中的代码行。请挑出有关_animator 的几行代码，将其添加到你的代码中。

这个脚本需要引用 Animator 组件。然后在 animator 上设置值(浮点型或布尔类型)。代码中唯一有点不明显的地方是在设置 Jumping 的布尔值之前的条件(_contact != null)。该条件可以防止 animator 中游戏开始时播放 jump 动画。虽然在技术上角色是一瞬间降落，但角色第一次接触到地面前不会有任何碰撞数据。

现在就实现了玩家的移动和动画，这个很不错的第三人称移动示例带有摄像机相对控制和角色动画播放。

8.5　小结

- 第三人称视角意味着摄像机跟随角色移动而不是在角色内。
- 模拟阴影改善了图形，类似于实时阴影和光照贴图。
- 控制是相对于摄像机而不是相对于角色的。
- 可以通过投射向下的光线来改善 Unity 的地面检测。
- 通过 Unity 的动画控制器设置高级动画，造就了栩栩如生的角色。

第 9 章

在游戏中添加交互设施和物件

本章涵盖：

- 编写程序，让玩家可以打开门
- 使用物理模拟使堆叠的箱子分散
- 创建可收集的物件以供玩家存储在仓库中
- 使用代码管理游戏状态，比如仓库数据
- 装备及使用仓库里的物件

实现功能物件是即将涉及的话题。之前的章节包含了完整游戏应有的一系列不同的元素：移动、敌人、用户界面等。但是我们的项目缺乏与敌人之外的其他交互，以及它们在游戏中的各种状态。本章将学习如何创建像门这种具有功能的物件。也会讨论如何收集物件，其中包括如何在当前关卡中和对象交互以及跟踪游戏状态。游戏通常需要跟踪一些状态，诸如玩家的当前状态、对象进展等。玩家的仓库就是这种状态的一个示例，所以需要创建一个代码架构，能用来跟踪玩家收集的物品。本章最后将建立一个动态的空间，使它像一个真正的游戏！

首先探索玩家单击按键来操作的设施(如门)，之后编写代码检测玩家在关卡中何时碰撞对象，并支持互动功能，例如推动附近的对象或收集可存储的物件。然后需要创建一个健壮的MVC格式代码框架来管理收集的仓库数据。最终，为游戏玩法编写接口以便使用仓库，例如可以用来打开门的钥匙。

警告　之前的章节相互之间非常独立，在技术上没有需要互相引用的地方。但是本章的代码清单要编辑第 8 章的一些脚本。如果直接开始阅读本章内容，那么为了完成本章项目，请从第 8 章下载示例项目。

这个示例项目中，有随机散落在该关卡中的各种设施和物件，优秀的游戏中会有很多精心设计的物件，但是对于仅用于测试功能的关卡而言，不需要如此仔细的准备。尽管如此，仍需要一些随意设置的对象，本章开头列出了要实现的设施和物件的顺序。

和往常一样，我们会解释一步步构建的代码，如果想要查看完成的所有代码，可以下载示例项目。

9.1　创建门和其他设施

虽然游戏中的关卡大都由静态的墙壁和风景组成，但它们也通常包括了很多具有功能的设施。现在讨论的对象是玩家可以与之交互和操作的设施，例如可以打开的灯或者旋转的风扇。设施之间可以有很大的差别，而这点仅受用户想象力的限制，但是几乎所有设施都使用相同类型的代码，让玩家激活设施。本章将实现两三个示例，然后调整这些代码，使它们能用于其他种类的设施。

9.1.1　用按键控制开关的门

我们创建的第一种设施将是一个可以打开和关闭的门，通过单击按键来操作门。在游戏中有很多不同种类的设施，操作它们的方法也不一样。这里介绍两种不同的设施，但门是游戏中最常见的互动设施，通过按键来使用物件也是最直截了当的方法。

在这个场景中，墙和墙之间有一些间隔，所以需要布置一个新的对象来挡住这个间隔。创建一个立方体对象，将它的变换设置为位置(2.5, 1.5, 17)，大小为(5, 3, 0.5)。在图 9-1 中可以看到所创建的这个门。

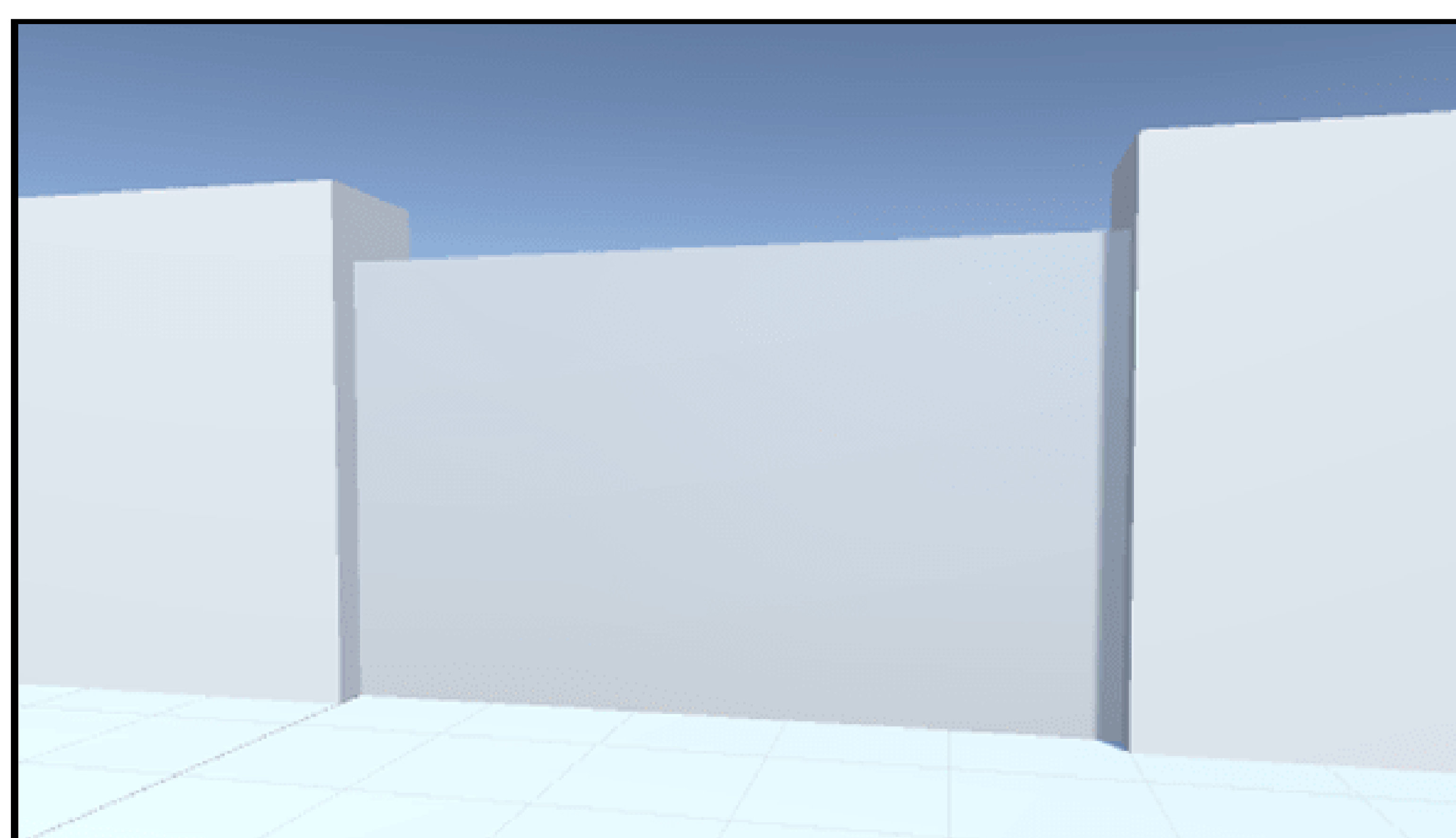


图 9-1　嵌入墙内的门对象

创建一个 C#脚本，命名为 DoorOpenDevice，然后将这个脚本放到门对象中。这

段代码(如代码清单 9.1 所示)会使对象表现出门的操作。

代码清单 9.1　根据命令打开和关闭门的脚本

```
using UnityEngine;
using System.Collections;

public class DoorOpenDevice : MonoBehaviour {
  [SerializeField] private Vector3 dPos;

  private bool _open;

  public void Operate() {
    if (_open) {
      Vector3 pos = transform.position - dPos;
      transform.position = pos;
    } else {
      Vector3 pos = transform.position + dPos;
      transform.position = pos;
    }
    _open = !_open;
  }
}
```

打开门时要偏移到的位置

布尔参数追踪门的状态

根据门的状态决定打开或关闭门

其中第一个变量 dPos 定义的是当门打开时它的偏移量。当门打开时，门会移动这个偏移量，当门关上时，会减去这个偏移量。第二个变量_open 是一个私有的布尔变量，用于追踪门是打开还是关闭状态。在 Operate()方法中，对象的变换设置为一个新位置，增加或者减少偏移量取决于门是否已经打开，然后打开或者关闭_open。

和其他序列化的变量一样，dPos 也显示在 Inspector 中。但这是一个 Vector3 值，所以以前只有一个输入框，这里会有三个，都在同一个变量的名下。输入门打开时的相对位置，为了让这个门下滑打开，这里的位移是(0, −2.9, 0)。因为门对象的高度是 3，向下移动 2.9 就可以在地板上留下门缝。

注意　立刻应用这个变换，但是当门打开时最好能够看到门的运动。如第 3 章所述，可以利用 tween 使对象平滑地移动。在不同的语境中 tween 的含义不同，在游戏编程中，它指的是使对象移动的代码。附录 D 提及了 Unity 的缓动系统。

现在，其他的代码需要引用 Operate()来打开或关闭门(该函数可以控制这两个操作)。目前还没有其他作用于玩家的脚本，下一步将编写这样的脚本。

9.1.2　在开门之前检查距离和朝向

创建一个新脚本并命名为 DeviceOperator。代码清单 9.2 会实现一个控制键，用来操作附近的设施。

代码清单 9.2　玩家的设施控制键

```
using UnityEngine;
using System.Collections;

public class DeviceOperator : MonoBehaviour {
  public float radius = 1.5f;

  void Update() {
    if (Input.GetButtonDown("Fire3")) {
      Collider[] hitColliders =
          Physics.OverlapSphere(transform.position, radius);
      foreach (Collider hitCollider in hitColliders) {
        hitCollider.SendMessage("Operate",
          SendMessageOptions.DontRequireReceiver);
      }
    }
  }
}
```

玩家激活设施的距离

响应 Unity 的输入设置中定义的输入按钮

OverlapSphere()返回一个附近对象的列表

SendMessage()尝试调用指定的函数，不管目标对象的类型

这段脚本中的大部分代码看起来都非常熟悉，但是代码中心有一个非常关键的新方法。首先，确定一个值，即距离多远可以操作设施。然后，在 Update()方法中，检查键盘输入，因为 Jump 键已经在 RelativeMovement 脚本中使用过，这次对 Fire3 做出响应(Fire3 在项目的输入设置中定义为左 Shift 键)。

现在分析这个关键的新方法：OverlapSphere()。该方法返回在给定位置的给定距离中所有对象的数组。通过传入玩家的位置以及 radius 变量，可以检测出玩家附近的所有对象。这个代码清单处理的对象可以各种各样(比如引爆一个炸弹，然后引用一个爆破力值)，但在当前情况下，我们试图对周围所有对象都调用 Operate()方法。

这个方法通过 SendMessage()调用，在之前的章节中，UI 按钮也使用了这种方法。和之前一样，使用 SendMessage()是因为我们不知道目标对象的确切类型，而这个命令可以作用于所有的 GameObject。这次将 DontRequireReceiver 选项传给这个方法，这是因为通过 OverlapSphere()返回的对象大部分是没有 Operate()方法的。通常，当对象中没有接受消息的组件时，SendMessage()会打印错误消息，但是在这里，这个错误消息不需要被关注，因为大部分对象会忽略这个消息。

编写完这段代码，就可以将这个脚本附加到玩家对象上。现在玩家可以站在门的附近，然后按下键来开门或者关门了。

这里有一个可以修复的小细节。目前而言，玩家的朝向并不影响门的开关，只要玩家离门足够近。也可以调整脚本，只操作玩家面对的设施，现在完成这个操作。在第 8 章中，可以通过计算点积来判断玩家的朝向，这是在两个向量上完成的数学运算，它会返回一个在-1 和 1 之间的值，其中 1 表示两个向量朝着完全相同的方向，而-1 表示它们的方向刚好相反。代码清单 9.3 给出了 DeviceOperator 脚本中的新代码。

代码清单 9.3　调整 DeviceOperator，只操作面向玩家的设施

```
...
foreach (Collider hitCollider in hitColliders) {
  Vector3 direction = hitCollider.transform.position - transform.position;
  if (Vector3.Dot(transform.forward, direction.normalized) > .5f) {   ← 当面向正确的方向时才发送消息
    hitCollider.SendMessage("Operate",
        SendMessageOptions.DontRequireReceiver);
  }
}
...
```

在使用点积法之前需要判断一下方向，即从玩家到物品的方向。通过在玩家位置和物品位置间做一个减法，就可以得到一个方向向量。然后在该方向向量和玩家目前的正朝向之间调用 Vector3.Dot()。当点积非常接近 1 时(尤其是当代码发现这个值大于 0.5 时)，就意味着这两个向量所指向的方向非常接近。

通过这个调整，门不会在玩家朝向其他方向时被打开或者关闭了，即使玩家离门非常近。这个方法可以应用到任意种类的设施的操作上。为了证明其灵活性，下面创建另一个示例设施。

9.1.3　创建一个变色监控器

前面创建了一个可以打开或者关上的门，同样的设施操作逻辑也可以运用在其他种类的设施上。接下来将创建另一种设施，它采用同样的方法来操作。这次创建一个显示在墙上的变色监控器。

建立一个新的立方体并放置它，使它在墙上突出一小部分。例如，选择位置(10.9, 1.5, −5)，然后创建一个新脚本，命名为 ColorChangeDevice，将这段脚本(如代码清单 9.4 所示)附加到墙上的显示器上。现在跑到墙壁的监控器处，单击和用于门一样的 operate 按键，显示器将改变颜色。如图 9-2 所示。

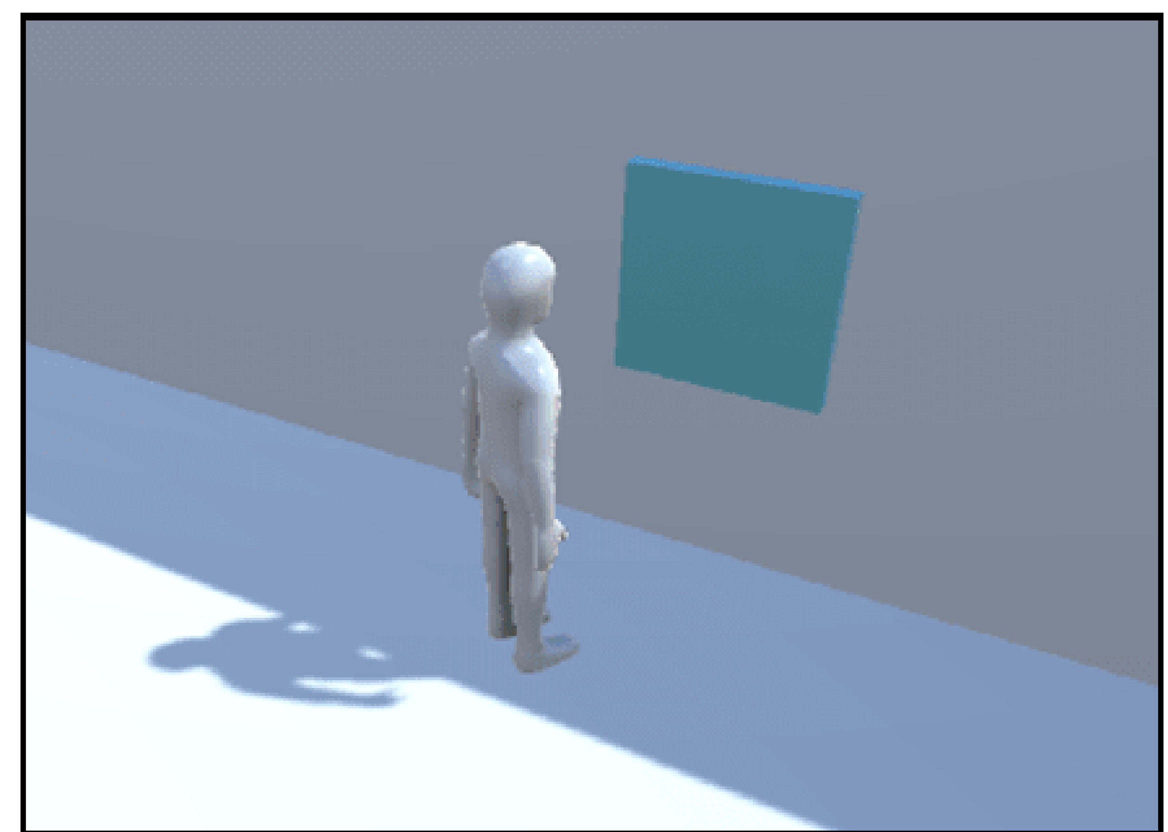


图 9-2　嵌入墙内的变色显示器

代码清单 9.4　能改变颜色的设施的脚本

```
using UnityEngine;
using System.Collections;

public class ColorChangeDevice : MonoBehaviour {
  public void Operate() {          ← 定义一个和门脚本同名的方法
    Color random = new Color(Random.Range(0f,1f),
        Random.Range(0f,1f), Random.Range(0f,1f));   ← 数字是介于 0 和 1 之间的 RGB 值
```

```
        GetComponent<Renderer>().material.color = random;
    }
}
```

设置对象上附加的材质的颜色

首先，声明一个与门脚本 DoorOpenDevice 中使用的同名函数 Operate，Operate 是设施操作脚本中使用的函数名，所以为了触发显示器设施，需要使用 Operate 这个函数名。在这个函数中，代码会给对象材质分配一个随机的颜色(要记住，颜色并不是对象本身的一个属性，而是对象所拥有的材质，材质才有颜色)。

注意 即使颜色通过红、蓝、绿三种成分来定义，这是大部分计算机图形中的标准，但在 Unity 的 Color 对象中，颜色的值是在 0 和 1 之间，而不是在大部分情况下都通用的 0 到 255(包括 Unity 的颜色拾取器 UI)。

前面讲解了一种在游戏中和设施交互的方法，实现了几种不同设备来进行演示。另一种和对象交互的方法是和对象碰撞，接下来讲解这个方法。

9.2　通过碰撞与对象交互

在上一节中，设施的操作是通过玩家敲击键盘来进行的，但这并不是玩家和当前关卡中的物件交互的唯一方式。另一个直接的方法就是响应玩家和对象的碰撞。Unity 将碰撞检测和物理设置内置于游戏引擎中，完成了大部分工作。虽然 Unity 会检测碰撞，但还需要编程来响应与对象的碰撞。

下面讲解三种在游戏中常见的碰撞响应：

- 推开并且倒下
- 触发关卡中的设施
- 接触后消失(适用于捡起物品时)

9.2.1　和具有物理功能的障碍物碰撞

首先，创建一堆箱子，然后在玩家撞入这堆箱子时使其分散开。尽管这一过程涉及的物理计算非常复杂，但 Unity 内置了所有的运算，并以非常逼真的方式将箱子分散开。

Unity 默认情况下并不会使用其物理模拟来移动对象。这个功能的实现需要向对象添加一个 Rigidbody 组件。这个概念最先在第 3 章中讨论过，因为敌人的火球也需要一个 Rigidbody 组件。同第 3 章所述，Unity 的物理系统仅作用于拥有 Rigidbody 组件的对象。单击 Add Component，然后在 Physics 菜单下找到 Rigidbody。

创建一个新的立方体对象，给它添加一个 Rigidbody 组件。创建若干个这样的立方体，把它们摆成一堆。在下面的示例中，创建了五个箱子，将它们堆成两层(如

图 9-3 所示)。

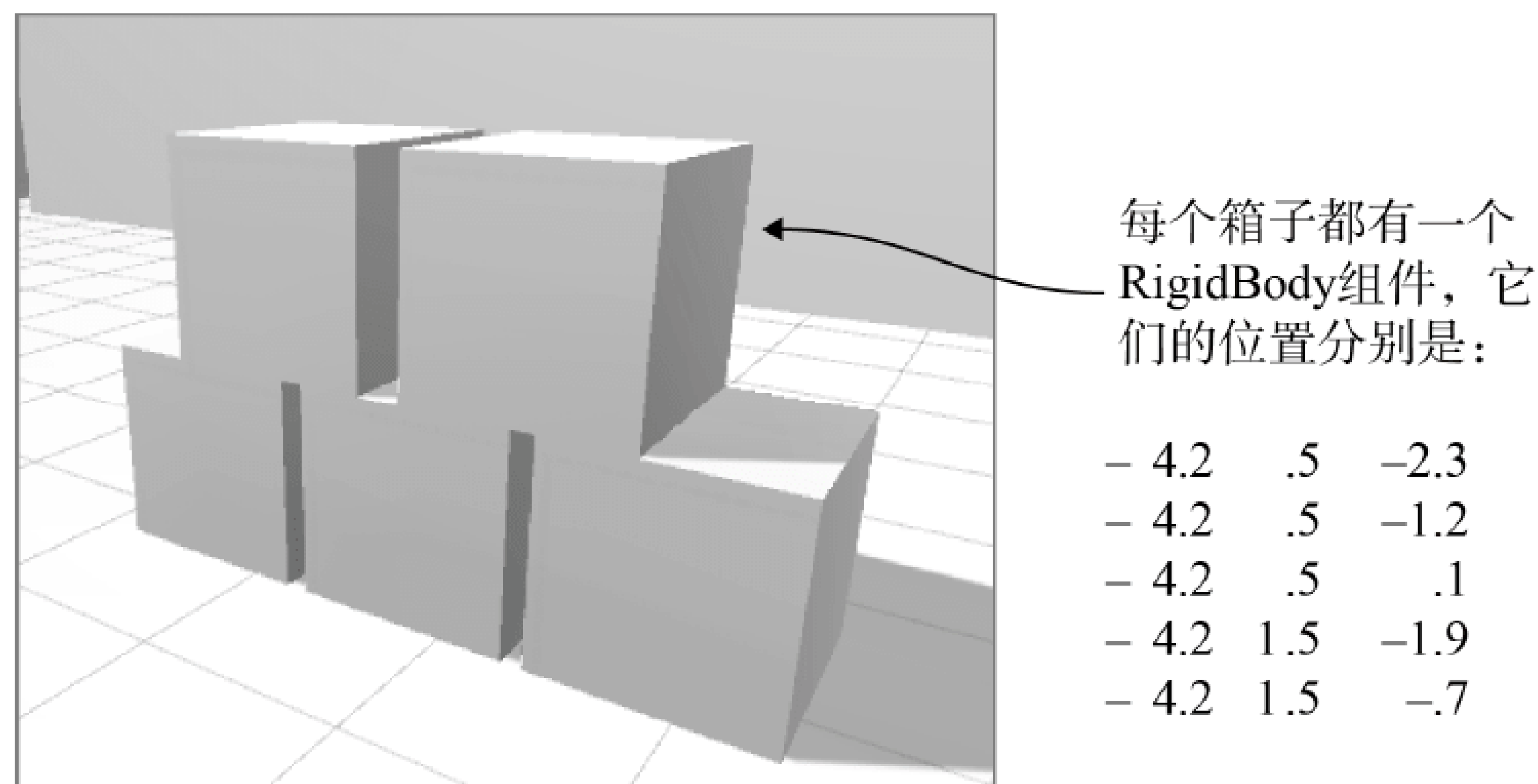


图 9-3　五个堆砌的箱子，用于碰撞

这些箱子现在准备好响应物理上的外力，让玩家在箱子上施加一个力，向玩家的 RelativeMovement 脚本中添加一小段代码，如代码清单 9.5 所示(这段代码就是在之前章节中的脚本)。

代码清单 9.5　为 RelativeMovement 脚本添加物理上的外力

```
...
public float pushForce = 3.0f;
...
void OnControllerColliderHit(ControllerColliderHit hit) {
   _contact = hit;

   Rigidbody body = hit.collider.attachedRigidbody;
   if (body != null && !body.isKinematic) {
     body.velocity = hit.moveDirection * pushForce;
   }
}
...
```

要应用的力量值

检查碰撞对象是否有 Rigidbody，以便接受物理上的外力

将速度应用到物理对象上

这段代码并没有太多的解释：无论玩家何时碰撞到对象，都检查碰撞到的对象是否有 Rigidbody 组件，如果有，给这个 Rigidbody 施加一个速度。

运行游戏，让玩家撞入箱子堆中，它们应该被撞散，非常逼真。这就是对场景中一堆箱子进行物理仿真所需要的操作！Unity 内置了物理仿真，所以不需要编写太多代码。这个模拟可以让对象在响应碰撞时四处移动，另一个可能的响应则是激活触发器事件。下面用这些触发器事件控制门。

9.2.2　用触发器对象操作门

之前通过按键操控门，现在，门的开和关将通过响应角色和场景中另一个对象的碰撞来完成。创建另一个门，将它放在另一个墙和墙的间隔中[这里复制了之前的那个门，把新门移动到(−2.5, 1.5, −17)的位置上]。现在，创建一个新的立方体用于这个触

发器对象，选中碰撞器的 Is Trigger 复选框。另外，在 Inspector 的右上角有一个 Layer 菜单，将触发器对象的层设置为 Ignore Raycast。最后，需要关掉这个对象的投射阴影(请记住，在选择对象时，这个设置在 Mesh Renderer 的下面)。

警告 这些细微但很重要的步骤很容易被遗漏：将对象用作触发器时一定要打开 Is Trigger。在 Inspector 中，检查一下 Collider 组件里的复选框。另外，将层改为 Ignore Raycast，这样触发器对象不会在光线投射中出现。

注意 第 3 章首次介绍触发器对象时，需要给这些对象添加Rigidbody 组件。但此时，Rigidbody 对于触发器对象而言不是必需的，因为触发器会对玩家做出响应(相较于早些时和墙壁的碰撞)。为了让触发器工作，无论是触发器还是进入触发器的对象，都需要启用 Unity 的物理系统，Rigidbody 组件满足这一要求，玩家的 CharacterController 也满足这个要求。

调整触发器对象的位置和大小，使其既覆盖到门，也覆盖到门附近的区域，选择位置为(−2.5, 1.5, −17)(门也是这个位置)，大小为(7.5, 3, 0.6)。另外，还需要将一个半透明的材质分配给这个对象，以直观地从实体对象中区分出触发器。使用 Assets 菜单创建一个新的材质，在 Project 视图中选择这个新建的材质。查看 Inspector，顶部的设置为 Rendering Mode(当前设置为默认值 Opaque)，在这个菜单中选择 Transparent。

现在，单击其色板，弹出 Color Picker 窗口。在该窗口的主要部分选择绿色，然后使用底部的滑块降低 alpha。从 Project 中将该材质拖动到对象上，图 9-4 显示了选择这种材质的触发器。

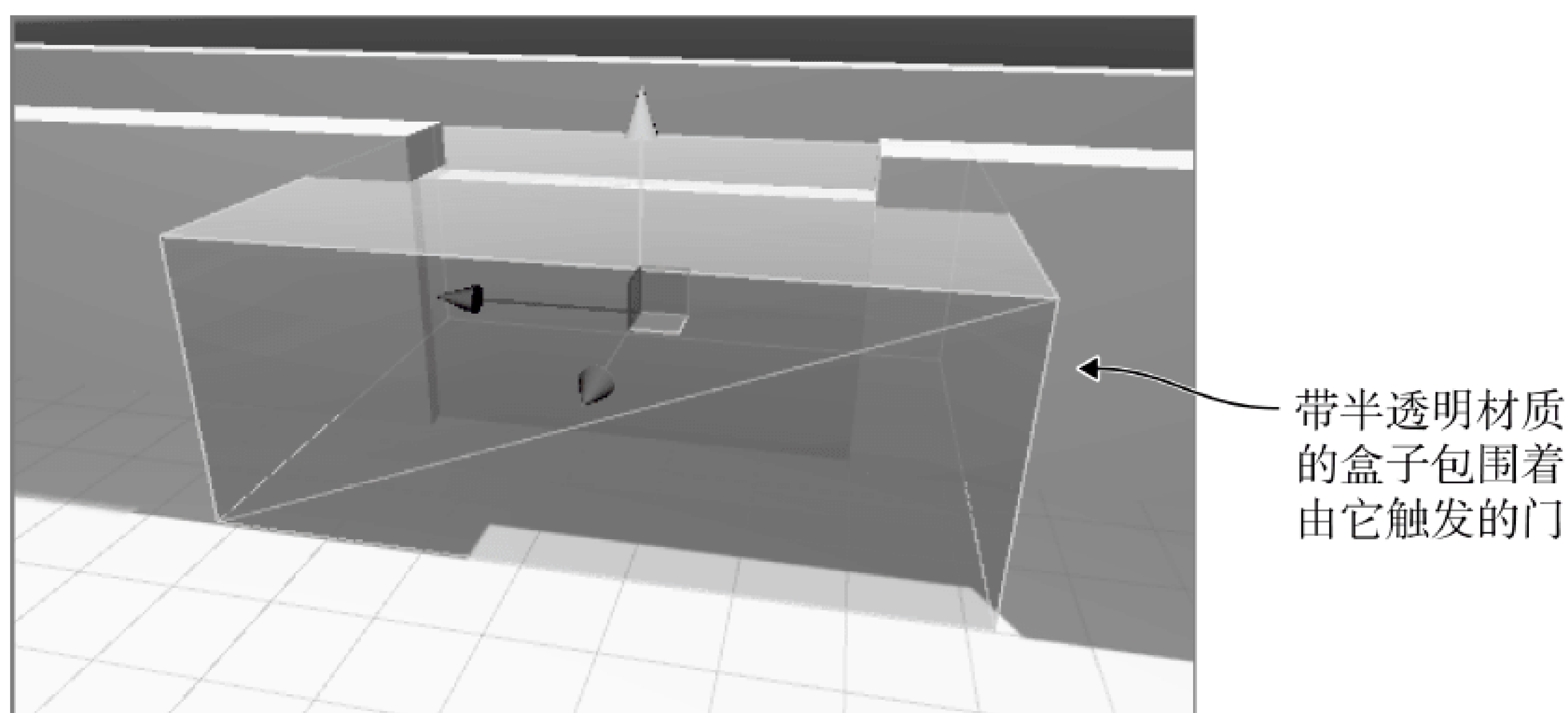


图 9-4　包含要触发的门的触发器

定义 触发器通常定义成体积，而不是对象，以便从概念上把实体对象和可穿透对象区分开。

运行游戏，现在可以自由地穿过触发器，Unity 依然记录与对象的碰撞，但是这些碰撞不再会影响到玩家的移动。为了响应这些碰撞，需要编写一些代码。具体来说，

希望这个触发器可以控制门。创建一段新脚本，命名为 DeviceTrigger(如代码清单 9.6 所示)。

代码清单 9.6　控制设施的触发器的代码

```
using UnityEngine;
using System.Collections;

public class DeviceTrigger : MonoBehaviour {
   [SerializeField] private GameObject[] targets;

   void OnTriggerEnter(Collider other) {
      foreach (GameObject target in targets) {
         target.SendMessage("Activate");
      }
   }

   void OnTriggerExit(Collider other) {
      foreach (GameObject target in targets) {
         target.SendMessage("Deactivate");
      }
   }
}
```

触发器要激活的目标对象列表

当另一个对象进入触发空间时，调用 OnTriggerEnter()

而当一个对象离开触发空间时，调用 OnTriggerExit()

这段代码为触发器定义了一个目标对象数组，尽管当前该列表中只有一个元素，但它为单一触发器控制多个设施提供了可能。可以遍历目标数组，向所有目标发送消息。这个循环发生在 OnTriggerEnter()和 OnTriggerExit()方法中。当另一个对象首次进入和离开触发器时，会调用这些函数(而不是在对象处于触发空间内时不断地调用这些函数)。

注意，和以前发送的信息不同的是，现在需要给门定义 Activate()和 Deactivate()函数。现在，给门脚本添加代码清单 9.7 所示的代码。

代码清单 9.7　将激活和取消激活函数添加到 DoorOpenDevice 脚本

```
...
public void Activate() {
   if (!_open) {
      Vector3 pos = transform.position + dPos;
      transform.position = pos;
      _open = true;
   }
}
public void Deactivate() {
   if (_open) {
      Vector3 pos = transform.position - dPos;
      transform.position = pos;
      _open = false;
   }
}
...
```

当门没有打开时，才打开

当门没有关闭时，才关闭

新的 Activate()和 Deactivate()方法的代码和之前的 Operate()代码几乎相同，但现在，开门和关门是用不同的函数处理的，而过去是用一个函数来完成这两个操作。

在所有需要的代码都到位之后，现在就可以使用触发空间来开关门了。将 DeviceTrigger 脚本添加到触发空间，然后将门和脚本中的 targets 属性关联起来。在 Inspector 中，首先设置数组的大小，然后将对象从 Hierarchy 视图中拖到目标数组中的槽里。因为只有一个门是用这个触发器控制的，所以在数组的 Size 中输入 1，然后将门拖动到目标槽里。

在完成这些之后，运行游戏，观察当玩家走向门和远离门时会发生什么。可以看到，当玩家走进和离开触发空间时，门会自动地打开和关闭。

这是另一个在游戏关卡中加入互动的好方法！这个触发空间方法不仅可以用于门这种设施上，也可以用这种方法收集对象。

9.2.3　收集当前关卡散落的物件

许多游戏包含一些可由游戏玩家捡起的物件，这些物件包括装备、血量包和物品。捡起这些物品时产生的基本碰撞机制是比较简单的，比较复杂的情况是发生在物件被捡起来之后，后面会介绍这一点。

创建一个球体对象，将它放在一个开阔场景大概中间高度的位置。缩小该对象，大概为(0.5, 0.5, 0.5)，除此之外，像处理大的触发体一样来处理这个球体。选择碰撞器中的 Is Trigger 设置，将对象的层设置为 Ignore Raycast，然后创建一个新材质，给予对象完全不同的颜色。因为这个对象非常小，不需要把它设置为半透明，所以不要把 alpha 滑块完全滑到底部。另外如第 8 章所述，对象中有关闭阴影投射的设置，而是否使用阴影取决于个人的主观判断，但是对于这种非常小的可拾取对象，最好关闭它。

现在，场景中的对象已经准备就绪，创建一个新脚本，将它附加到那个对象上。脚本命名为 CollectibleItem，如代码清单 9.8 所示。

代码清单 9.8　在与玩家接触时，删除物品的代码

```
using UnityEngine;
using System.Collections;

public class CollectibleItem : MonoBehaviour {
   [SerializeField] private string itemName;

   void OnTriggerEnter(Collider other) {
      Debug.Log("Item collected: " + itemName);
      Destroy(this.gameObject);
   }
}
```

在 Inspector 中输入这个物品的名称

这段脚本非常简短，给这个对象指定 name 值，这样在场景中就可以有不同的对象。OnTriggerEnter()用于销毁自身，现在还在控制台上显示调试消息，但是最终它会

被有用的代码代替。

警告　请确保在 this.gameObject 而不是 this 上调用 Destroy()！不要把这两者弄混淆，this 只引用这个脚本组件，而 this.gameObject 引用这段脚本所附加的对象。

回到 Unity，代码中添加的变量应该在 Inspector 中可见。输入一个名称来区分这个对象。将所创建的第一个对象命名为 energy，然后复制这个对象若干次，再逐个更改副本的名称。还要创建 ore、health 和 key(这些名称都必须准确，因为它们要在后面的代码中用到)。给每个对象创建独立的材质，以便给它们赋予不同的颜色。给 energy 用的是蓝色，ore 用的是深灰色，health 用的是粉色，key 用的是黄色。

提示　与这里给对象命名不同的是，在很多复杂的游戏中，对象通常都有一个标识符用来查找更多的数据。例如，一个对象可能分配的 id 是 301，然后 id301 所关联的信息可能包含显示名称、图片、描述等。

给这些对象创建预制件，然后就可以在整个游戏关卡中克隆它们。第 3 章解释过，将对象从 Hierarchy 视图拖到 Project 视图中，可以将对象变成一个预制件，现在对这四个对象都进行这样的操作。

注意　在 Hierarchy 列表中，对象的名称会变为蓝色，蓝色的名称表示对象是预制件的实例。右击一个预制件的实例，选择 Select Prefab，然后选择对象是其实例的预制件。

将这些预制件的实例拖出，放在游戏关卡中较开阔的区域，甚至可以拖动同一个对象的多个副本来进行测试。运行游戏，然后走到对象附近去“收集”它们。这看起来相当简单。但是当捡起一个对象时，什么都没有发生。下面要跟踪每个被收集的对象，为此，需要建立一个仓库代码结构。

9.3　管理仓库数据和游戏状态

现在已经编写了收集物品的特性，接下来需要为游戏的仓库编写后台数据管理(类似网页编码模式)。要编写的代码非常类似于大多数 Web 应用程序中的 MVC 架构。MVC 架构的优点是将数据存储从显示在屏幕的对象中解耦出来，更便于试验和交互开发。甚至当数据和/或显示比较复杂时，使用 MVC 都可以使程序中某一部分的修改不影响其他部分。

也就是说，这种结构在不同游戏中各不相同。不是每个游戏的数据管理需求都是相同的，例如，角色扮演游戏会有高度的数据管理需求，因此需要实现类似 MVC 的架构。然而益智游戏不怎么需要数据管理，因此不必为数据的管理构建复杂的解耦结

构。游戏状态能通过场景中特定的控制对象来跟踪(实际上，前面章节介绍了如何处理游戏状态)。

在这个项目中，需要管理玩家的仓库。接下来开始创建仓库需要的结构。

9.3.1 设置玩家和仓库管理器

这里主要的目标是将所有的数据管理任务分割成独立的、良好定义的模块，每个模块只负责管理自己的区域。我们要创建单独的模块，使用 PlayerManager 保存玩家的状态(比如玩家的血量)，使用 InventoryManager 保存玩家的物品清单。这些数据管理器就像 MVC 模式中的 Model，在大多数场景中，控制器是一个不可见的对象(这里并不需要控制器，但是回想一下前面章节提到的 SceneController)，余下的场景类似于 MVC 模型中的 View(视图)。

这里有一个更高级别的“管理器的管理器(manager of managers)”来跟踪所有的独立模块。除了保存所有管理器的清单之外，这个更高级别的管理器还控制各个管理器的生命周期，特别是在最初时初始化它们。游戏中所有其他的脚本通过这个主管理器都可以访问这些集中式的模块。具体来说，其他代码可以用主管理器中的一些静态属性来连接所需的某个模块。

访问集中式共享模块的设计模式

多年来，涌现了许多设计模式，它们都旨在解决将程序的不同部分连接到整个程序共享的集中式模块的问题。

但是很多软件工程师并不喜欢单例模式，因此他们采用了其他选择，比如服务定位器和依赖注入。本书代码采用的模式是在静态变量的简洁性和服务定位器的灵活性之间的一种折中。

这种设计既保留了代码的易用性，又允许在不同模块中切换。例如，使用单例模式请求 InventoryManager 总是会引用同一个类，因此将代码与这个类紧密关联起来。另一方面，从服务定位器中访问 Inventory，可以返回 InventoryManager 或 DifferentInventoryManager。有时，能够在相同模块的不同版本中来回切换(比如，将游戏部署到不同的平台)是很方便的。

为了让主管理器以一致的方式引用其他模块，这些模块必须继承一个共同的基类中的属性。为此要使用一个接口，许多编程语言(包括 C#)都允许定义一种其他类都要遵循的设计。PlayerManager 和 InventoryManager 将实现一个共同的接口(在此称为 IGameManager)，然后主 Managers 对象将把 PlayerManager 和 InventoryManager 都视为 IGameManager 类型。图 9-5 演示了这种设置。

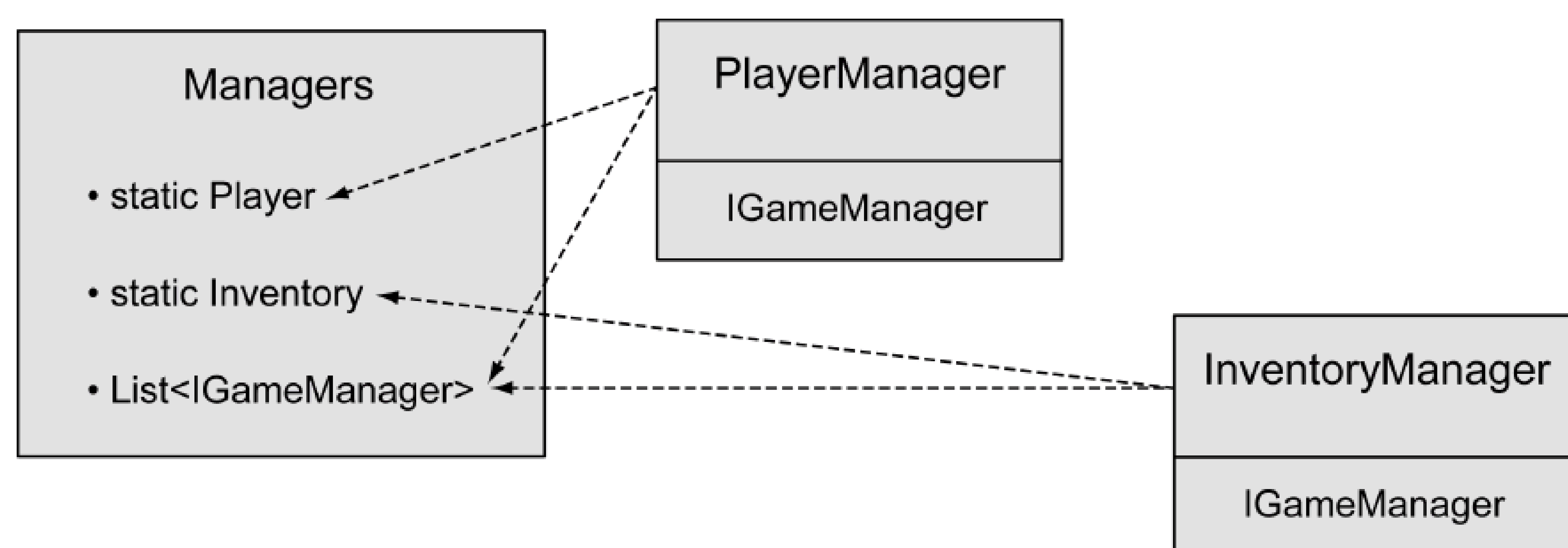


图 9-5　模块图以及它们之间的关系

尽管前面讨论的所有代码架构都由后台上不可见的模块组成，但 Unity 仍然需要把脚本连接到场景中的对象上来运行这些代码。就像在之前的项目中对具体场景的控制器所做的事情一样，我们将创建一个空的 GameObject 对象来关联这些数据管理器。

9.3.2　编程实现游戏管理器

以上解释了需要理解的所有概念，下面该编写代码了。新建一个称为 IGameManager 的脚本(见代码清单 9.9)。

代码清单 9.9　数据管理器将实现的基本接口

```
public interfaceIGameManager {
    ManagerStatu sstatus {get;}        ← 这是一个需要定义的枚举

    void Startup();
}
```

嗯，这个文件几乎没有任何代码。注意它甚至没有继承 MonoBehaviour，接口本身不执行任何操作，它存在的意义仅仅是提供其他类上的结构。这个接口声明了一个属性(一个拥有 getter 函数的变量)和一个方法，它们都要在实现这个接口的类中实现。status 属性告诉其他代码，这个模块是否完成了初始化。Startup()方法的目的是处理管理器的初始化，初始化的任务在该方法中完成，这个方法还会设置管理器的状态。

注意，该属性的类型是 ManagerStatus，这是一个还未编写的枚举，因此创建脚本 ManagerStatus.cs(见代码清单 9.10)。

代码清单 9.10　ManagerStatus: IGameManager 所有可能的状态

```
public enum ManagerStatus{
   Shutdown,
   Initializing,
   Started
}
```

这是另外一个几乎没有任何代码的文件。这次列出了管理器所有可能的状态，因此 status 属性只能是上面列出的几种状态之一。

现在 IGameManager 已经编写完毕，可以在其他脚本中实现它。代码清单 9.11 和代码清单 9.12 包含了 PlayerManager 和 InventoryManager 的代码。

代码清单 9.11　InventoryManager

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class InventoryManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  public void Startup() {
    Debug.Log("Inventory manager starting...");
    status = ManagerStatus.Started;
  }
}
```

导入新的数据结构(代码清单 9.14 中会用到)

属性可以从任何地方获取，但只能在这个脚本中设置

任何长时间运行的任务都放在这里

如果是长时间运行的任务，状态变为‘Initializing’

代码清单 9.12　PlayerManager

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class PlayerManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  public int health {get; private set;}
  public int maxHealth {get; private set;}

  public void Startup() {
    Debug.Log("Player manager starting...");

    health = 50;
    maxHealth = 100;

    status = ManagerStatus.Started;
  }

  public void ChangeHealth(int value) {
    health += value;
    if (health > maxHealth) {
      health = maxHealth;
    } else if (health < 0) {
      health = 0;
    }

    Debug.Log("Health: " + health + "/" + maxHealth);
  }
}
```

继承一个类，实现一个接口

可以使用保存的数据初始化这些值

其他脚本不能直接设置血量，但是可以调用这个方法

现在，InventoryManager 是一个以后填充的 shell，而 PlayerManager 具有这个项

目所需的所有功能。这些管理器都继承自类 MonoBehaviour 且实现了 IGameManager 接口。这意味着，所有的管理器既获得了 MonoBehaviour 的功能又实现了 IGameManager 定义的结构。IGameManager 的结构是一个属性和一个方法，所有管理器都定义了它们。

定义 status 属性，这样可以从任何地方获取状态(getter 方法是公有方法)，但是只能在脚本内设置状态(setter 方法是私有方法)。IGameManager 接口中的方法是 Startup()，两个管理器都定义了这个方法。在两个管理器中，初始化会立即完成(InventoryManager 没有执行任何操作，而 PlayerManager 设置了一组值)，因此将 status 设置成 Started。但是数据模块的初始化中可能有一些长时间运行的任务(比如加载保存的数据)，在初始化时，Startup()将运行这些任务，将管理器的状态设置为 Initializing。这些任务完成后，将 status 改为 Started。

现在终于可以将所有一切和主管理器的管理器连接起来了。再创建一个脚本，称为 Managers(见代码清单 9.13)。

代码清单 9.13 管理器的管理器

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

[RequireComponent(typeof(PlayerManager))]        ← 确保存在不同的管理器
[RequireComponent(typeof(InventoryManager))]

public class Managers : MonoBehaviour {
   public static PlayerManager Player {get; private set;}      ← 其他代码用来访问管理器的静态属性
   public static InventoryManager Inventory {get; private set;}

   private List<IGameManager> _startSequence;      ← 启动时要遍历的管理器列表

   void Awake() {
      Player = GetComponent<PlayerManager>();
      Inventory = GetComponent<InventoryManager>();

      _startSequence = new List<IGameManager>();
      _startSequence.Add(Player);
      _startSequence.Add(Inventory);

      StartCoroutine(StartupManagers());      ← 异步启动序列
   }

   private IEnumerator StartupManagers() {
      foreach (IGameManager manager in _startSequence) {
      manager.Startup();
      }

      yield return null;

      int numModules = _startSequence.Count;
```

```
        int numReady = 0;

        while (numReady < numModules) {          ← 循环至所有管理
            int lastReady = numReady;               器都启动为止
            numReady = 0;

            foreach (IGameManager manager in _startSequence) {
                if (manager.status == ManagerStatus.Started) {
                    numReady++;
                }
            }

            if (numReady > lastReady)
                Debug.Log("Progress: " + numReady + "/" + numModules);
            yield return null;
        }                                          ← 再次检查之前，
                                                      停顿一帧
        Debug.Log("All managers started up");
    }
}
```

这个模式最重要的部分是顶部的静态属性。这些属性允许其他脚本通过 Managers.Player 或 Managers.Inventory 这样的语法来访问不同模块。这些属性初始是空的，但是当 Awake()方法中的代码运行时，它们就会被赋值。

提示 就像 Start()和 Update()，Awake()是 MonoBehaviour 自动提供的另一个方法。它与 Start()很相似，当代码第一次运行时，只运行一次。但是在 Unity 的代码执行序列中，Awake()比 Start()执行得更早，允许完成比任何其他代码模块都要先执行的初始化任务。

Awake()方法也列出了启动序列，然后启动协程来运行所有的管理器。具体来说，这个方法创建了一个 List 对象，然后用 List.Add()方法添加管理器。

定义 List 是 C#提供的一个集合数据结构。列表对象与数组很相似：它们以特定的类型声明，把一系列元素依次存入其中。但是 List 可以在创建之后改变大小，而数组一旦创建之后就不能改变大小。

因为所有的管理器都实现了 IGameManager，所以这段代码可以把它们都当成这个类型，并为每个管理器调用已定义的 Startup()方法。这个启动序列是作为协程运行的，因此它与其他的游戏处理模块(例如，在启动屏幕上的进度条)一起异步执行。

这个启动函数首先遍历整个管理器列表，并依次调用各自的 Startup()方法。然后它进入一个循环，检查管理器是否启动，直到它们都启动了为止。一旦所有的管理器都启动了，这个启动函数就会在最终完成前通知我们。

提示　之前编写的管理器的初始化过程非常简单且不需要等待，然而通常这个基于协程的启动序列可以优雅地处理长时间运行的异步启动任务，例如加载保存过的数据。

现在，所有的代码结构已经编写完毕。回到 Unity，创建一个空的 GameObject 对象。与往常一样，对于这种空的代码对象，将它放在(0, 0, 0)处，给这个对象赋予一个描述性的名称，比如 Game Managers。将脚本组件 Managers、PlayerManager 和 InventoryManager 关联到这个新的对象上。

现在运行游戏，场景中应该没有任何可见的变化，但是在控制台上可以看到一系列记录启动序列过程的日志信息。假设管理器正确启动，下面就该编写仓库管理器了。

9.3.3　把物品存储在集合对象中：List 与 Dictionary

收集到的物品列表可以存储在 List 对象中。代码清单 9.14 演示了如何将物品列表添加到 InventoryManager 中。

代码清单 9.14　将物品添加到 InventoryManager

```
...
private List<string> _items;

public void Startup() {
   Debug.Log("Inventory manager starting...");

   _items==new List<string>();          ← 初始化空的物品列表

   status = ManagerStatus.Started;
}
                                                打印当前仓库的
private void DisplayItems() {                ←  控制台消息
   string itemDisplay = "Items: ";
   foreach (string item in _items) {
      itemDisplay += item + " ";
   }
   Debug.Log(itemDisplay);
}
                                             其他脚本不能直接操作物品列表，
public void AddItem(string name) {        ←  但是可以调用这个方法
   _items.Add(name);

   DisplayItems();
}
...
```

对于 InventoryManager 有两个关键的修改：首先，添加了一个 List 对象来存放物品；其次，添加了一个其他代码都可以调用的公有方法 AddItem()。AddItem()方法将物品添加到列表，然后将列表信息打印到控制台。现在，在 CollectibleItem 脚本中做一些小小的调整，来调用这个新的 AddItem()方法(见代码清单 9.15)。

代码清单 9.15　在 CollectibleItem 中使用新的 InventoryManager

```
...
void OnTriggerEnter(Collider other) {
   Managers.Inventory.AddItem(name);
   Destroy(this.gameObject);
}
...
```

现在，在运行收集物品的代码时，可以在控制台信息中看到仓库在增大。这非常酷，但暴露了 List 数据结构的一个缺陷：当收集同类型的多个物品(比如收集第二个 Health 项)时，会列出两个副本，而不是累计同类型的所有物品(如图 9-6 所示)。根据游戏的不同，仓库也许要分别跟踪每件物品，但是在大多数游戏中，仓库应该累计同一物品的多个副本。这可以使用 List 来实现，但是使用 Dictionary 更方便、有效。

Items: energy health ore health energy ore key
UnityEngine.Debug:Log(Object)

图 9-6　同一物品被打印多次的控制台信息

定义 Dictionary 是 C#提供的另一个集合数据结构。字典中的条目通过标识符(或者键)来访问，而不是通过它们在列表中的位置来访问。它类似于哈希表，但更灵活，因为字典中的键在字面上可以是任意类型(例如，“返回该 GameObject 的条目”)。

修改 InventoryManager 中的代码，以使用 Dictionary 替代 List。使用代码清单 9.16 中的代码替换代码清单 9.14 中的代码。

代码清单 9.16　InventoryManager 中的物品字典

```
...
private Dictionary<string, int> _items;      // Dictionary 由两种类型声明：键和值

public void Startup() {
   Debug.Log("Inventorymanager starting...");

   _items = new Dictionary<string, int>();

   status = ManagerStatus.Started;
}

private void DisplayItems() {
   string itemDisplay = "Items: ";
   foreach (KeyValuePair<string, int> item in _items) {
      itemDisplay += item.Key + "(" + item.Value + ") ";
   }
   Debug.Log(itemDisplay);
}

public void AddItem(string name) {
   if (_items.ContainsKey(name)) {      // 在输入新数据之前，检查已有的项
```

```
        _items[name] += 1;
      } else {
        _items[name] = 1;
      }

      DisplayItems();
    }
    ...
```

所有代码与之前看起来一样，但有一些微妙的区别。如果对 Dictionary 数据结构不是很熟悉，请注意它是由两个类型声明的。List 只由一种类型(列入列表的值的类型)声明，而 Dictionary 声明了键(即标识符)和值的类型。

在 AddItem()方法中还存在一个逻辑。在每个条目加入列表之前，需要检查 Dictionary 是否已经包含了该条目，这就是 ContainsKey()方法的作用。如果是一个新条目，就把计数器置为 1。但如果这个条目已经存在，就递增它的数量。

运行新的代码，仓库信息就包含了每个条目的数量，如图 9-7 所示。

Items: energy(1) health(2) ore(2) key(1)
UnityEngine.Debug:Log(Object)

图 9-7　多个相同条目聚合后的控制台信息

最后，玩家仓库中收集的物品得到了管理！这看起来像是用大量代码处理一个相对简单的问题，而且，如果这是最终的目标，它就被过度设计了。然而，精心设计的代码架构的意义在于，把所有数据放在独立、灵活的模块中，当游戏变得复杂时，这是一个非常有用的模式。例如，现在可以编写 UI 显示，分离的代码块也更容易维护。

9.4　使用和装备物品的仓库 UI

在游戏中，可以以多种方式使用仓库中的物品集合，但是所有的用法都依赖于某种仓库 UI，这样玩家可以看到他们收集的物品。然后，一旦仓库显现在玩家面前，就可以编写 UI 交互程序，允许玩家单击物品。下面将编写两个具体的示例(包括钥匙和消费血量包)，之后就可以将这段代码作用在其他类型的物品上。

注意　如第 7 章所述，Unity 既有立即模式的旧 GUI，又有基于精灵的新 UI 系统。本章采用立即模式的 GUI，因为这种系统能更快地实现，需要的设置更少。更少的设置对于实践练习极有好处。然而，基于精灵的 UI 系统更优雅，但对于实际的游戏，应使用更优雅的接口。

9.4.1　在 UI 中显示仓库物品

要在 UI 中显示物品，首先需要给 InventoryManager 添加两个方法。现在物品列表是私有的，只能在管理器内部访问，为了显示这个列表，这个信息中必须包含访问

数据的公有方法。将代码清单 9.17 中的两个方法添加到 InventoryManager 中。

代码清单 9.17 将数据访问方法添加到 InventoryManager 中

```
...
public List<string> GetItemList() {            返回所有 Dictionary
   List<string> list = new List<string>(_items.Keys);   键的列表
   return list;
}

public int GetItemCount(string name){          返回仓库中
   if (_items.ContainsKey(name)) {              物品的个数
      return _items[name];
   }
   return 0;
}
...
```

GetItemList()方法返回仓库中的物品列表。你可能会想：等等，难道我们不是花了很大的代价才将 List 转为仓库对象吗？这里的区别在于，每种物品仅在列表中出现一次。如果仓库中存在两个血量包，例如，health 只会在列表中出现一次。这是因为 List 是由 Dictionary 中的键(而不是每个独立的物品)产生的。

GetItemCount()方法返回了仓库中指定物品的数量。例如，调用 GetItemCount(“health”)来询问“仓库中有多少血量包？”，这样，UI 可以显示每个物品及其数量。

给 InventoryManager 添加了这些方法后，就可以创建 UI 显示。下面在屏幕顶部水平显示所有的物品。这些物品将以图标的形式显示，所以需要将这些图片导入到项目中。如果所有的资源都放在 Resources 目录下，Unity 就以一种特殊的方式来处理这些资源。

提示 放在 Resources 目录中的资源可以通过 Resources.Load()方法来加载。否则，资源只能通过 Unity 的编辑器放到场景中。

图 9-8 展示了四个图标图片和放置这些图片的目录结构。创建 Resources 目录，然后在 Resources 目录中创建 Icons 目录。

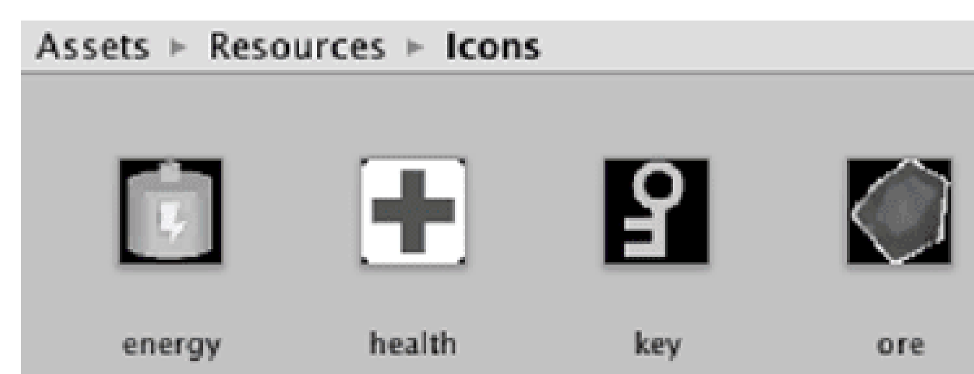


图 9-8 Resources 目录中存放的装备图标的图片资源

图标都已设置完毕，现在创建一个名为 Controller 的空游戏对象，然后将一个新脚本 BasicUI 赋给它(见代码清单 9.18)。

代码清单 9.18 显示仓库的 BasicUI

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
```

```
public class BasicUI : MonoBehaviour {
  void OnGUI() {
    int posX = 10;
    int posY = 10;
    int width = 100;
    int height = 30;
    int buffer = 10;

    List<string> itemList = Managers.Inventory.GetItemList();
    if (itemList.Count == 0) {
      GUI.Box(new Rect(posX, posY, width, height), "No Items");
    }
    foreach (string item in itemList) {
      int count = Managers.Inventory.GetItemCount(item);
      Texture2D image=  Resources.Load<Texture2D>("Icons/"+item);
      GUI.Box(new Rect(posX, posY, width, height),
           new GUIContent("(" + count + ")", image));
      posX += width+buffer;
    }
  }
}
```

当仓库为空时，显示一条消息

循环中每次向一边偏移

从 Resources 目录中加载资源的方法

代码清单 9.18 以水平的方式显示所收集的物品(见图 9-9)及其数量。如第 3 章所述，每个 MonoBehaviour 自动响应 OnGUI()方法。在渲染 3D 场景之后，这个方法在每一帧中都会执行。

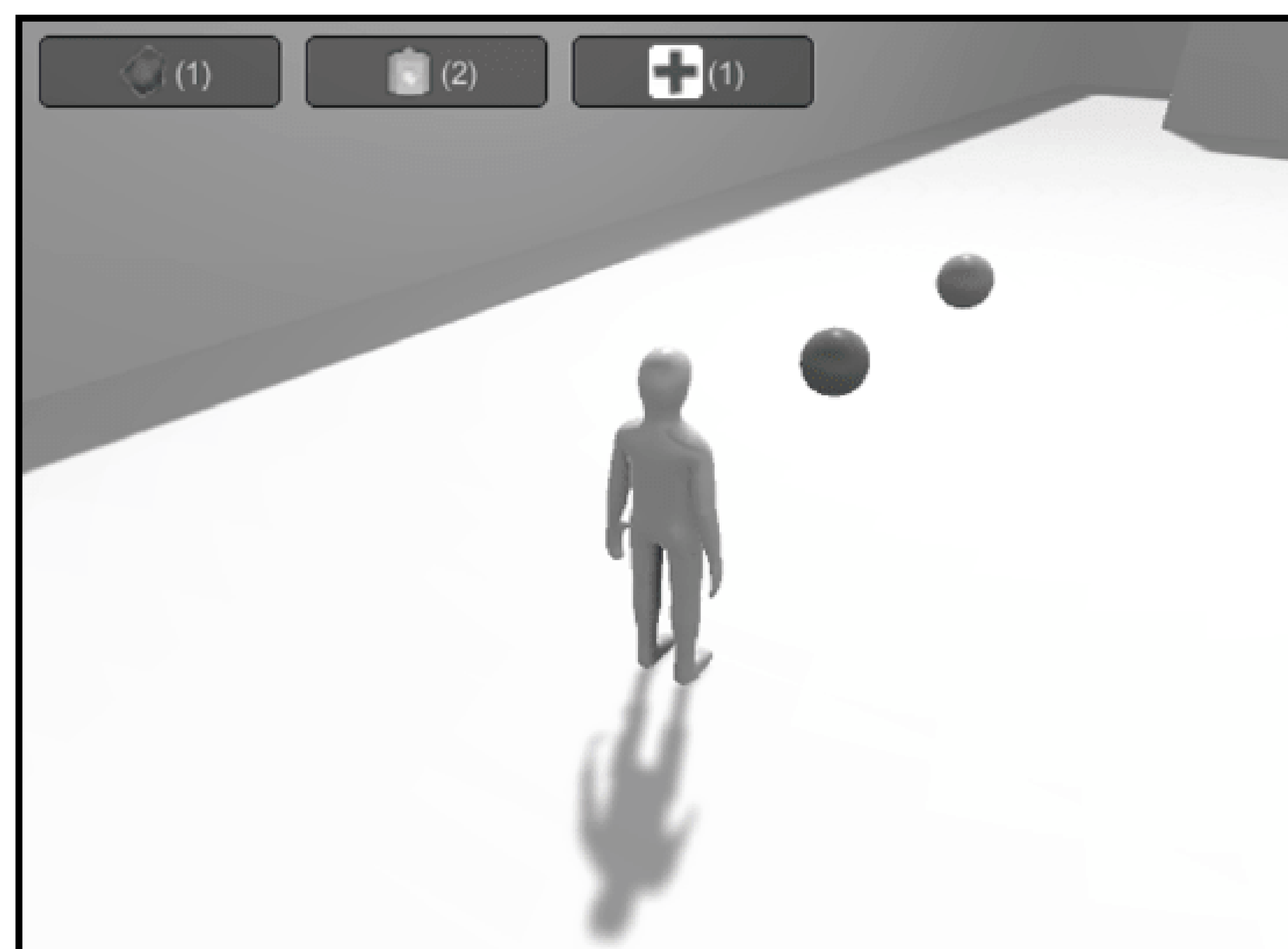


图 9-9　显示仓库的 UI

在 OnGUI()中，首先定义一组标记 UI 元素位置的值。当遍历所有的物品时，这些值会递增，从而将 UI 元素定位在一行。具体的 UI 元素通过 GUI.Box 绘制，这些显示在盒子中的文本和图片是不可交互的。

Resources.Load()方法用于从 Resources 目录中加载资源。该方法便于根据名称来加载资源，注意物品的名称将作为参数。必须指定要加载的类型，否则，这个方法的返回值就是通用对象。

UI 显示了收集的物品。现在可以使用这些物品了。

9.4.2　装备一个用来开门的钥匙

下面介绍两个关于使用仓库物品的示例，这些示例可以推广至任何物品类型。第一个示例是装备一个用来打开门的钥匙。

此时，DeviceTrigger 脚本并没有关注物品(因为这个脚本是在仓库代码之前编写的)。代码清单 9.19 展示了如何调整该脚本。

代码清单 9.19　在 DeviceTrigger 中需要一把钥匙

```
...
public bool requireKey;

void OnTriggerEnter(Collider other) {
   if (requireKey && Managers.Inventory.equippedItem != "key") {
      return;
   }
...
```

可以看到，脚本需要一个新的公有变量和一个查找钥匙是否已装备的条件。布尔参数 requireKey 在 Inspector 中显示为复选框，这样可以通过一些触发器获得钥匙，而不是通过其他方式获得。OnTriggerEnter()开头的条件检查 InventoryManager 中是否已装备了钥匙，这需要将代码清单 9.20 添加到 InventoryManager 中。

代码清单 9.20　InventoryManager 的装备物品代码

```
...
public string equippedItem {get; private set;}
...
public bool EquipItem(string name) {
   if (_items.ContainsKey(name) && equippedItem !=   name){   ◄── 检查仓库中有该物品，但还没有被装备
     equippedItem = name;
     Debug.Log("Equipped " + name);
     return true;
   }

   equippedItem = null;
   Debug.Log("Unequipped");
   return false;
}
...
```

在代码清单的顶部添加了被其他代码检查的 equippedItem 属性，然后添加了公有方法 EquipItem()，允许其他代码改变被装备的物品。这个方法装备还未装备的物品，或卸下已装备的物品。

最后，为了让玩家装备物品，需要把这个功能添加到 UI 上。为此，代码清单 9.21

将添加一行按钮。

代码清单 9.21　添加给 BasicUI 的装备功能

```
...
    foreach (string item in itemList) {          ← 斜体代码在脚本中已存在，显示在此处仅作为参考

      int count = Managers.Inventory.GetItemCount(item);
      GUI.Box(new Rect(posX, posY, width, height), item +
            "(" + count + ")");
      posX += width+buffer;
    }

    string equipped = Managers.Inventory.equippedItem;
    if (equipped != null) {                       ← 显示当前装备的物品
    posX = Screen.width - (width+buffer);
    Texture2D image = Resources.Load("Icons/"+equipped) as Texture2D;
    GUI.Box(new Rect(posX, posY, width, height),
            new GUIContent("Equipped", image));
  }

  posX = 10;
  posY += height+buffer;

  foreach (string item in itemList) {             ← 遍历所有物品来创建按钮
    if (GUI.Button(new Rect(posX, posY, width, height),
            "Equip "+item)) {                     ← 如果单击按钮，则运行其包含的代码
      Managers.Inventory.EquipItem(item);
    }
    posX += width+buffer;
  }
 }
}
```

为了显示装备的物品，再次使用了 GUI.Box()。但这个元素是非交互式的，所以这一行的 Equip 按钮使用 GUI.Button()绘制。这个方法创建了一个按钮，当单击该按钮时，会执行 if 语句中的代码。

所有代码都完成后，在 DeviceTrigger 中选择 requireKey 选项，然后运行游戏。试着在装备钥匙之前跑进触发空间，什么都没发生。现在收集一个钥匙，然后单击按钮装备它，跑入触发空间，门会打开。

下面的操作纯粹是为了打趣：可以在位置(-11, 5, -14)上放置一把钥匙，简单增加游戏玩法的难度，看看能否拿到那把钥匙。下面学习如何使用血量包。

9.4.3　通过使用血量包来恢复玩家的血量

使用物品来恢复玩家的血量是另一个有用的常见示例。这需要修改两处代码：一处是 InventryManager 中增加一个新方法，另一处是 UI 中增加一个新按钮(分别见代码清单 9.22 和代码清单 9.23)。

代码清单 9.22　InventoryManager 中的新方法

```
...
public bool ConsumeItem(string name) {
   if (_items.ContainsKey(name)) {          检查物品是否在仓库中
      _items[name]--;
      if (_items[name] == 0) {              如果数量减为 0，则移除物品
        _items.Remove(name);

      }
   }else {                                  如果仓库中没有该物品，则给出响应
      Debug.Log("cannotconsume " + name);
      return false;
   }

   DisplayItems();
   return true;
}
...
```

代码清单 9.23　给 BasicUI 添加一个血量物品

```
...                                                  斜体代码在脚本中已存在，显示在此处仅便于参考
        foreach (string item in itemList),
          if (GUI.Button(new Rect(posX, posY, width,height),
                   "Equip "+item)){
              Managers.Inventory.EquipItem(item);
          }
                                                     新代码的开始处
          if (item == "health") {
             if (GUI.Button(new Rect(posX, posY + height+buffer, width,
                          height), "Use Health")) {   如果单击按钮，就会运行其包含的代码
                Managers.Inventory.ConsumeItem("health");
                Managers.Player.ChangeHealth(25);
             }
          }

          posX += width+buffer;
       }
   }
}
```

这个新的 ConsumeItem()方法正好与 AddItem()方法的作用相反，它在仓库中检查某个物品，如果找到该物品，就递减它的数量。它必须考虑一些微妙的场景，比如物品数量减到 0 的情况。UI 代码会调用这个新的仓库方法，还调用 PlayerManager 从一开始就拥有的 ChangeHealth()方法。

如果收集了一些血量物品并使用了它们，血量信息就显示在控制台上。至此，就完成了使用仓库物品的多个示例。

9.5　小结

- 按键和碰撞触发器都可以用来操作设施。
- 物理对象可以响应碰撞力或触发空间。
- 复杂的游戏状态可以通过特殊的对象来全局性地访问。
- 可以在 List 或 Dictionary 数据结构中组织对象的集合。
- 跟踪物品的装备状态可能会影响游戏的其他部分。

第Ⅲ部分

冲 刺 阶 段

现在，我们掌握了 Unity 的很多知识。知道如何编写玩家的控件，如何创建到处游走的敌人，如何将交互设施添加到游戏中，甚至知道如何使用 2D 和 3D 图形构建游戏！这些内容几乎是开发完整游戏所需要知道的所有知识点，但不是全部知识点。我们还需要完成最后几个任务，诸如将音频加入游戏中，将已完成的分散内容整合在一起。

第 10 章

将游戏连接到互联网

本章涵盖：

- 为天空生成动态视觉效果
- 在协程中使用 Web 请求下载数据
- 解析诸如 XML 和 JSON 等常见的数据格式
- 显示从互联网下载的图像
- 将数据发送到 Web 服务器

本章将学习如何通过网络发送和接收数据。前面章节构建的项目代表了各种不同的游戏类型，但那些项目都使玩家之间相互隔离。连接到互联网并交换数据对于所有的游戏类型都变得越来越重要。很多游戏几乎完全通过互联网进行，与其他玩家社区保持连接，这种游戏通常称为 MMO(massively multiplayer online，大型多人在线)，最广为人知的是 MMORPG(MMO role-playing games，大型多人在线角色扮演)。甚至当游戏不需要保持持续不断的连接时，现代的视频游戏通常也会包括一些特性，例如将分数报告到全球高分列表中，或分析记录以帮助改进游戏。Unity 提供了上述网络支持，接下来将探讨这些特性。

Unity 支持多种方式的网络通信，因为不同的方式适用于不同的需求。然而，本章主要介绍最常用的互联网通信：发出 HTTP 请求。

什么是 HTTP 请求？

假设大多数读者知道什么是 HTTP 请求，这里简单介绍：HTTP 是用于向 Web 服务器发送请求和接收响应的通信协议。例如，当单击网页上的链接时，浏览器(客户端)将请求发送到指定地址，接着服务器使用新页面响应。可以将 HTTP 请求设置为不同的方法，特别是设置为 GET 或 POST 方法，以获取或发送数据。

HTTP 请求是可靠的，因此大多数互联网应用都围绕它们而创建。这种请求和处理这种请求的基础设施被设计得很健壮，能够处理网络中的各种错误。

一个好的类比是，想象现代单页面 Web 应用程序的工作原理(与之相对的是基于服务器端生成 Web 页面的老式 Web 开发)。在基于 HTTP 请求构建的在线游戏中，Unity 中开发的项目本质上是一个采用 Ajax 风格与服务器通信的胖客户端。熟悉这种方法可能会误导有经验的 Web 开发人员。电子游戏通常比 Web 应用程序有更严格的性能要求，这些差异会影响设计决策。

警告 Web 应用程序和视频游戏中时间的衡量不同。更新一个网站花费半秒看起来很短，但在一个高强度动作游戏中暂停半秒的时间都是一种折磨。“快速”这个概念是根据情况定义的。

在线游戏通常连接到该游戏特定的服务器。为了便于学习，我们连接到一些免费可用的互联网数据源，包括天气数据和可以下载的图像。本章最后部分要求设置一个自定义 Web 服务器，这部分内容虽然是可选的，但本章依然会介绍如何通过开源软件用简单的方式来实现它。

本章计划介绍 HTTP 请求的多种用法，以便学习它们在 Unity 中的工作原理：

(1) 创建户外场景(特别是，构建一个可以响应天气数据的天空)

(2) 编写代码，从互联网请求天气数据

(3) 解析响应并基于数据修改场景

(4) 从互联网下载并显示图像

(5) 将数据发送到服务器(本例中是天气日志)

用于本章项目的示例游戏并不重要。本章的所有内容都是将新脚本添加到已有的项目中，且不修改任何现有的代码。对于示例代码，采用第 2 章的移动示例，主要为了可以在数据改变时以第一人称视角观察天空。本章项目和玩法没有直接的关系，但显然，对于我们创建的大多数游戏，都要把它们连接到网络上(例如，基于服务器的响应而产生敌人)。

下面开始第一步！

10.1　创建户外场景

由于要下载天气数据，因此首先创建一个可以显示天气的户外场景。最复杂的部分是天空，但先花点时间将户外贴图应用到关卡的几何体上。

如第 4 章所述，从 www.textures.com 上获取一些图像，将这些图像应用到关卡的墙壁和地板。记住将下载图像的大小修改为 2 的 n 次幂，例如 256×256。接着将图像导入到 Unity 项目中，创建材质，并将图像赋予到材质上(也就是将图像拖动到材质的贴图槽)。将材质拖动到场景中的墙壁或地板上，接着增加材质的平铺数(尝试设置一个或两个方向的平铺数为 8 或 9)，以便图像不会以丑陋的方式拉伸。

一旦地板和墙壁处理完毕，就可以开始装饰天空。

10.1.1　使用天空盒生成天空视觉效果

如第 4 章所述，首先导入天空盒图像：在 www.93i.de 上下载天空盒图像。这次下载 DarkStormy 系列和 TropicalSunnyDay(这个项目中的天空将会更复杂)。将这些贴图导入到 Project 视图中，并(如第 4 章所述)将贴图的 Wrap Mode 设置为 Clamp。

现在创建用于这个天空盒的新材质。在这个材质设置的顶部，单击 Shader 菜单，查看可用的着色器(shader)列表。将鼠标移动到 Skybox 部分，并选择子菜单中的 6-Slided。激活这个着色器后，材质现在有 6 个贴图槽(而不像标准着色器只有一个小的 Albedo 贴图槽)。

将 SunnyDay 天空盒图像拖动到新材质的贴图槽中。图像名称和它们赋予的贴图槽名称相对应(top、front 等)。链接好 6 个贴图后，就可以将这个新材质用作场景的天空盒。

在 Lighting 窗口(Window | Lighting | Settings)中指定这个天空盒材质。将天空盒材质赋予到窗口顶部的 Skybox 槽(将材质拖动到 Skybox 槽上或者单击 Skybox 槽旁边的小圆圈按钮)。单击 Play，可以看到如图 10-1 所示的画面。

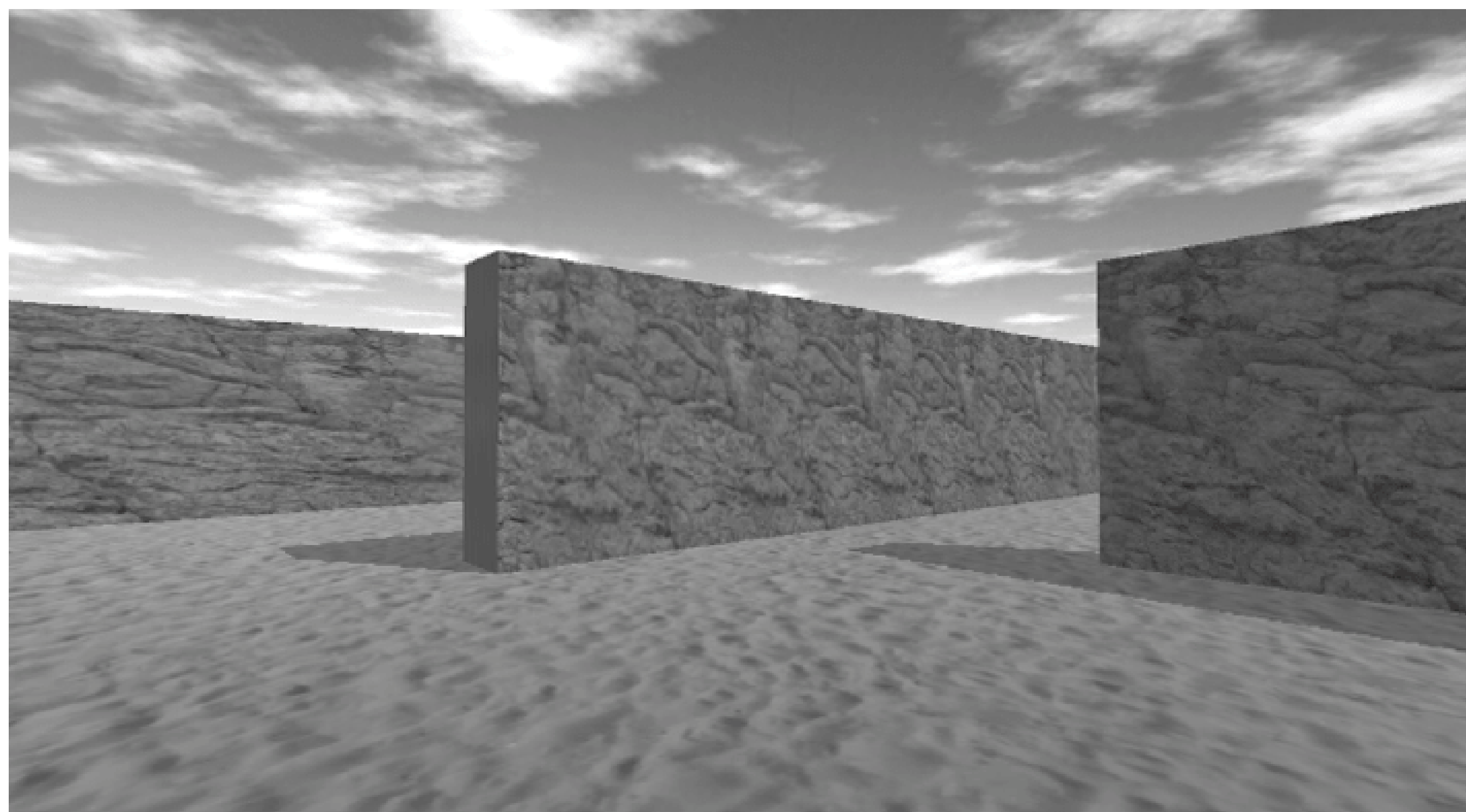


图 10-1　带有天空背景图像的场景

现在有了一个室外场景！天空盒可以很好地创建环绕玩家的大气环境。但 Unity 内置的天空盒着色器有一个明显的限制：天空盒材质的图像不能改变，这导致天空完全是静态的。接下来，通过创建自定义着色器来消除这个限制。

10.1.2　通过代码设置大气环境

TropicalSunnyDay 系列中的图像适合于晴天，但如果要在晴天和阴天之间变换，该怎么办？这将需要第二套天空图像(一些阴天的图片)，因此需要新的着色器实现天空盒。

如第 4 章所述，着色器是一个简短的程序，其中的指令用于渲染图像。这意味着可以编写新着色器，而事实上正是如此。接下来将创建新的着色器，使用两个天空盒图像集，并在它们之间变换。幸运的是，用于这个目的的着色器已经存在于 Unify Community 维基的脚本集合中：http://wiki.unity3d.com/index.php?title=SkyboxBlended。

在 Unity 中创建新着色器脚本：像创建 C#脚本一样，进入 Create 菜单，但选择 Standard Surface Shader。该资源命名为 SkyboxBlended，然后双击着色器，打开脚本。复制维基页面上的代码，粘贴到着色器脚本中。顶部一行声明了 Shader “Skybox/Blended”，这告诉 Unity 将新着色器添加到 Skybox 分类的着色器列表下(常规天空盒所在的分类)。

注意　接下来不探讨着色器程序的细节。Shader 编程是一个相当高级的计算机图形主题，超出了本书的讨论范围。如果在阅读完本书后想进一步学习，可以以 http://docs.unity3d.com/Manual/Shaders Overview.html 为起点。

现在可以将材质的着色器设置为 Skybox Blended。有 12 个贴图槽，即两组 6 个图像。将 TropicalSunnyDay 图像赋予前六张贴图，剩下的贴图使用 DarkStormy 系列天空盒图像。

这个新着色器也在设置的顶部附近添加了一个 Blend 滑动条。Blend 值控制了要显示多大的天空盒图像集。将滑动条从一端调整到另一端时，天空盒将从晴天变换到阴天。可以通过调整滑动条并运行游戏来进行测试，但当游戏运行时，手动调整天空没有什么作用，因此接下来编写变换天空的代码。

在场景中创建一个空对象，并命名为 Controller。创建一个新脚本并命名为 WeatherController。将脚本拖到空对象上，接着编写代码清单 10.1 中的代码。

代码清单 10.1　从晴天到阴天变换的 WeatherController 脚本

```
using UnityEngine;
using System.Collections;

public class WeatherController : MonoBehaviour {
   [SerializeField] private Material sky;
   [SerializeField] private Light sun;
```

引用 Project 视图中的材质，而不仅仅是场景中的对象

```
    private float _fullIntensity;

    private float _cloudValue = 0f;

    void Start() {
      _fullIntensity = sun.intensity;    // 初始灯光最开始为满强度
    }

    void Update() {
      SetOvercast(_cloudValue);
      _cloudValue += .005f;    // 为了持续变换，每帧增加值
    }

    private void SetOvercast(float value) {    // 同时调整材质的 Blend 值和灯光强度
      sky.SetFloat("_Blend", value);
      sun.intensity = _fullIntensity - (_fullIntensity * value);
    }
}
```

接下来指出这段代码中的一些要点，但关键的新方法是 SetFloat()，该方法在代码的底部。前面的其他代码都很熟悉，只有该方法的那一行是新代码。SetFloat()方法在材质上设置了一个数值。该方法的第一个参数指明了设置哪个值。在本例中，材质有一个称为 Blend 的属性(注意材质属性在代码中以下划线开头)。

代码的剩余部分定义了一些变量，包括材质和灯光。对于材质，需要引用刚刚创建的混合天空盒材质，但灯光如何处理呢？场景从晴天过渡到阴天时，灯光会变暗，随着 Blend 值的增加，将调低灯光的强度。场景中的平行光是主灯光，照亮任何地方，将平行光拖动到 Inspector 中。

注意 Unity 中的高级灯光系统引入天空盒的原因是为了实现更真实的效果。然而，这种照明方式不适用于变化的天空盒，因此要关闭它。在 Lighting 窗口的底部可以关闭 Auto generate 复选框，现在只有单击按钮，才会更新天空盒。将天空盒的 Blend 设置到一半，以获得平衡的视觉效果，接着单击 Auto 复选框旁边的按钮，手动烘焙光照贴图(灯光信息存储在以 Scene 命名的新文件夹中)。

当脚本开始运行时，它初始化灯光的强度。脚本将保存开始值，并认为这个开始值为“满”强度(译者注：“满”强度指运行时灯光最大的强度)。这个满强度会在以后脚本减弱灯光强度时使用。

接着代码在每一帧递增值，并使用该值调整天空。具体而言，代码每帧都调用 SetOvercast()，而该函数包括对场景进行多个调整。之前已经解释了 SetFloat()的作用，这里不再赘述，最后一行代码调整了灯光的强度。

现在运行场景，观察代码的运行。结果如图 10-2 所示：几秒后，场景从晴天过渡到阴天。

警告 Unity 的一个意想不到的问题是材质上对 Blend 的修改是永久的。Unity 在游戏停止运行时重置场景中的对象，但对直接从 Project 视图(例如天空盒材质)中关联的资源的修改却是永久的。这只发生在 Unity 编辑器中(在游戏部署到编辑器外部时，修改不会在运行游戏间传递)，如果忘记了这一点，则可能会产生令人沮丧的 bug。

看到场景从晴天过渡到阴天真的很炫酷。但真正的目标是：让游戏中的天气和真实世界的天气同步。为此，需要从互联网下载天气数据。

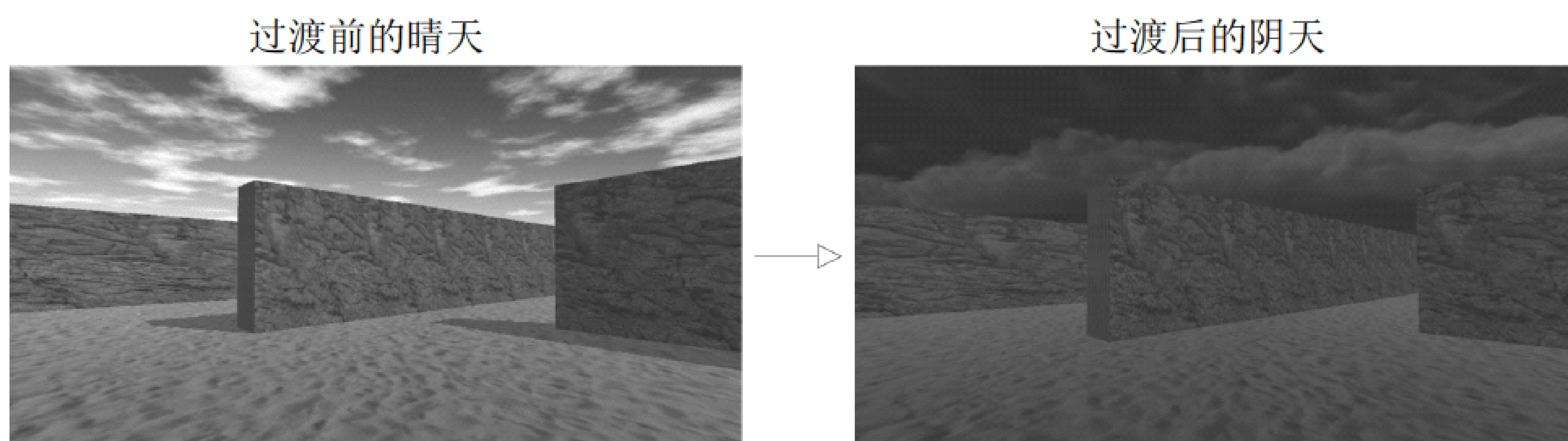


图 10-2　场景从晴天过渡到阴天

10.2　从互联网服务下载天气数据

现在已经设置好了户外场景，接下来编写代码，下载天气数据，并基于下载的数据修改场景。这个任务将提供一个使用 HTTP 请求获取数据的好例子。有许多 Web 服务都提供天气数据，其列表参见 www.programmableweb.com/。这里选择 OpenWeatherMap，代码示例使用其位于 http://openweathermap.org/api 的 API(应用程序编程接口，一种使用代码命令而不是图形界面访问其服务的方式)。

定义 Web 服务或 Web API 是一个连接到互联网并根据请求返回数据的服务器。Web API 和网站没有技术上的区别。网站是为网页返回数据的 Web 服务，且浏览器会拦截 HTML 数据，作为可视化文档。

注意 Web 服务经常要求注册，即使是免费服务也是如此。例如，如果进入 OpenWeatherMap 的 API 页面，它有获取 API 键的说明，这个值将粘贴到请求中。

接下来编写的代码结构和第 9 章的 Managers 架构一样。这次将编写 WeatherManager 类，它在中心主管理器中初始化。WeatherManager 负责天气数据的获取和保存，为了实现这个功能，它需要能和互联网通信。

为了实现联网，要创建一个称为 NetworkService 的工具类。NetworkService 将处

理连接到互联网并发出 HTTP 请求的细节。接着 WeatherManager 调用 NetworkService 进行请求并传回响应。图 10-3 展示这个代码结构的操作方式。

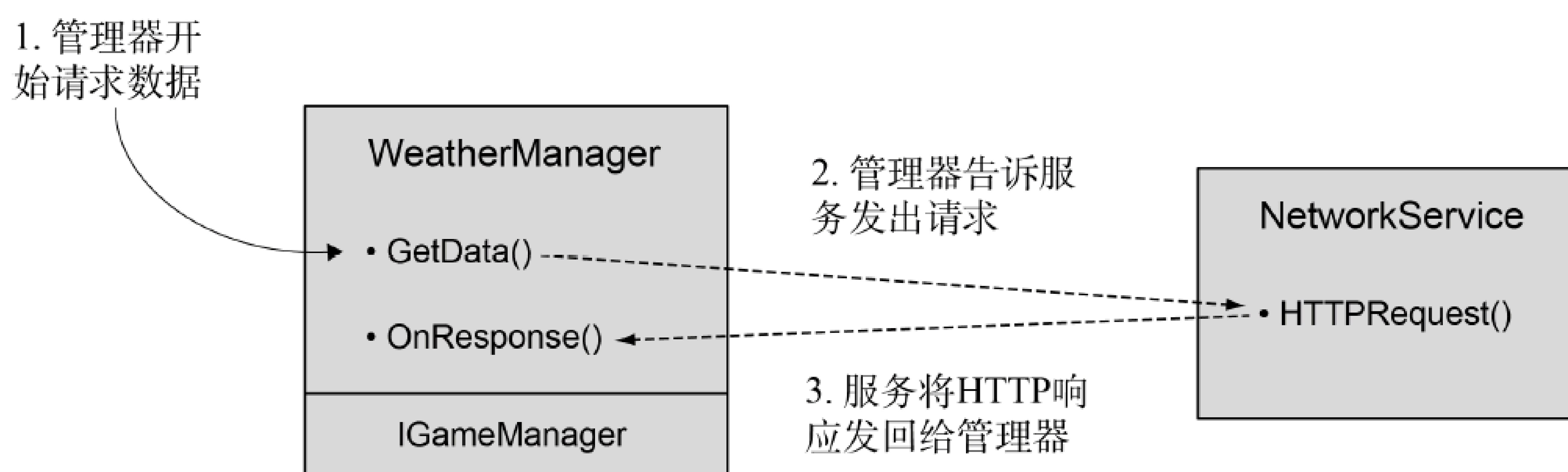


图 10-3　展示联网代码结构的图

WeatherManager 显然需要访问 NetworkService 对象。为此，应在 Managers 中创建对象，并当管理器初始化时，将 NetworkSerivce 对象注入不同的管理器中。这种方式不仅让 WeatherManager 拥有 NetworkService 的引用，而且后续创建的其他管理器也可以拥有 NetworkService 的引用。

为了引入第 9 章的 Managers 的代码架构，首先复制 ManagerStatus 和 IGameManager (记住 IGameManager 是所有管理器必须实现的接口，而 ManagerStatus 是 IGameManager 使用的枚举)。接下来需要稍微修改 IGameManager 来容纳新的 NetworkService 类，因此创建新脚本 NetworkSerivce(现在先让它空着，稍后再填充它)，接着如代码清单 10.2 所示调整 IGameManager。

代码清单 10.2　调整 IGameManager，以包含 NetworkService

```
public interface IGameManager {
  ManagerStatus status {get;}

  void Startup(NetworkService service);   ← Startup 函数现在带有一个参数：被注入的对象
}
```

接下来创建 WeatherManager 来实现这个稍微调整的接口。创建一个新的 C#脚本(如代码清单 10.3 所示)。

代码清单 10.3　WeatherManager 的初始化脚本

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class WeatherManager : MonoBehaviour, IGameManager {
    public ManagerStatus status {get; private set;}

   // Add cloud value here (listing 10.8)
   private NetworkService _network;
```

```
    public void Startup(NetworkService service) {
      Debug.Log("Weather manager starting...");

      _network = service;                      ← 保存注入的 NetworkService 对象

      status = ManagerStatus.Started;
    }
}
```

WeatherManager 目前实际还没有任何功能。现在它只包含实现 IGameManager 接口需要的最小代码量：声明接口所需的 status 属性，实现 Startup()函数。后续部分会填充这个空的框架。最后从第 9 章复制 Managers 并调整它，来启动 WeatherManager(如代码清单 10.4 所示)。

代码清单 10.4　Managers.cs 调整为初始化 WeatherManager

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
                                                      需要新的管理器而
[RequireComponent(typeof(WeatherManager))]   ←        不是玩家和仓库

public class Managers : MonoBehaviour {
   public static WeatherManager Weather {get; private set;}

   private List<IGameManager> _startSequence;

   void Awake() {
      Weather = GetComponent<WeatherManager>();

      _startSequence = new List<IGameManager>();
      _startSequence.Add(Weather);

   StartCoroutine(StartupManagers());
   }
                                                          实例化 NetworkService，
   private IEnumerator StartupManagers() {                以便注入到所有管理器中
      NetworkService network = new NetworkService();  ←

      foreach (IGameManager manager in _startSequence) {
         manager.Startup(network);           ←   启动时将 network
      }                                          服务传递给管理器

      yield return null;

      int numModules = _startSequence.Count;
      int numReady = 0;

      while (numReady < numModules) {
         int lastReady = numReady;
         numReady = 0;

         foreach (IGameManager manager in _startSequence) {
```

```
                if (manager.status == ManagerStatus.Started) {
                  numReady++;

                }
            }

            if (numReady > lastReady)
                Debug.Log("Progress: " + numReady + "/" + numModules);

            yield return null;
        }

        Debug.Log("All managers started up");
    }
}
```

上面的代码清单是 Managers 代码架构所需的代码。如前面章节所述，在场景中创建游戏管理器对象，并将 Managers 和 WeatherManager 附加到空对象上。尽管管理器目前还没做任何事，但可以在控制台看到正确设置后的启动消息。

现在，已经有了一些代码模板作为铺垫！接下来开始编写联网代码。

10.2.1　使用协程请求 HTTP 数据

NetworkService 当前是一个空的脚本，因此可以在该脚本中编写代码，创建 HTTP 请求。所需要了解的主要的类是 UnityWebRequest。Unity 提供的 UnityWebRequest 类用于与互联网进行通信。使用 URL 实例化请求对象会将请求发送给该 URL。

协程能和 UnityWebRequest 类一起工作，用于等待请求完成。协程在第 3 章中介绍过，那时使用协程让代码暂停一段时间。回想一下第 3 章中对协程的解释：协程是特殊的函数，它似乎在程序的后台周期性地运行，之后返回到程序的剩余部分继续执行。当与 StartCoroutine()方法一起使用时，yield 关键字将导致协程临时暂停，退出程序流，在下一帧继续执行。

在第 3 章中，在 WaitForSeconds()处产生了一个协程，WaitForSeconds()返回的对象让函数暂停执行数秒。发送请求时产生的协程将使函数暂停执行，直到网络请求完成。这里的程序流类似于 Web 应用程序中的异步 Ajax 调用：首先发送一个请求，接着继续执行剩下的程序，在一段时间后收到响应。

上述就是协程处理互联网请求的原理，接下来编写代码。

下面在代码中实现这一点。首先打开 NetworkService 脚本，使用代码清单10.5 中的内容替换默认模板。

代码清单 10.5　在 NetworkService 中发送 HTTP 请求

```
using UnityEngine;
using UnityEngine.Networking;
```

```
using System.Collections;
using System;                                         发送请求的 URL

public class NetworkService {
  private const string xmlApi =                       在 GET 模式下创建
"http://api.openweathermap.org/data/2.5/              UnityWebRequest 对象

    w eather?q=Chicago,us&mode=xml&APPID=<your api key>";

  private IEnumerator CallAPI(string url, Action<string> callback) {
    using (UnityWebRequest request = UnityWebRequest.Get(url)) {

     yield return request.Send();       下载时暂停

     if (request.isNetworkError) {                    在响应中
       Debug.LogError("network problem: " + request.error);   检查错误
     } else if (request.responseCode !=
          (long)System.Net.HttpStatusCode.OK) {
       Debug.LogError("response error: " + request.responseCode);
     } else {
      callback(request.downloadHandler.text);         可以像原始函数
     }                                                一样调用委托
    }
  }
  public IEnumerator GetWeatherXML(Action<string> callback) {
    return CallAPI(xmlApi, callback);
  }                                    通过相互调用的协程方法
}                                      调用产生级联
```

警告　Action 类型包含在 System 名称空间中，注意脚本顶部附加的 using 语句。不要忘记脚本中的这些细节！

回想一下前面解释过的代码设计：WeatherManager 将通知 NetworkService 获取数据。上述所有代码还不能运行，需要建立之后由 WeatherManager 调用的代码。为了研究该代码清单，下面先从底部开始往上阅读代码。

编写彼此嵌套的协程方法

GetWeatherXML()是一个在代码外部能告知 NetworkService 发出 HTTP 请求的协程方法。注意这个函数使用 IEnumerator 作为它的返回类型，协程中使用的方法必须把 IEnumerator 声明为其返回类型。

最初 GetWeatherXML()方法可能看起来会有点奇怪，它没有 yield 语句。yield 语句可以让协程暂停，这意味着每个协程必须在某个地方有 yield 语句。这证明 yield 语句可以通过多个方法嵌套返回。如果初始协程方法调用其他方法，而其他方法有部分代码返回 yield，那么协程将在第二个方法(包含 yield 代码的方法)的内部暂停和恢复。因此 CallAPI()中的 yield 语句暂停了在 GetWeatherXML()中启动的协程。图 10-4 显示了这个代码流。

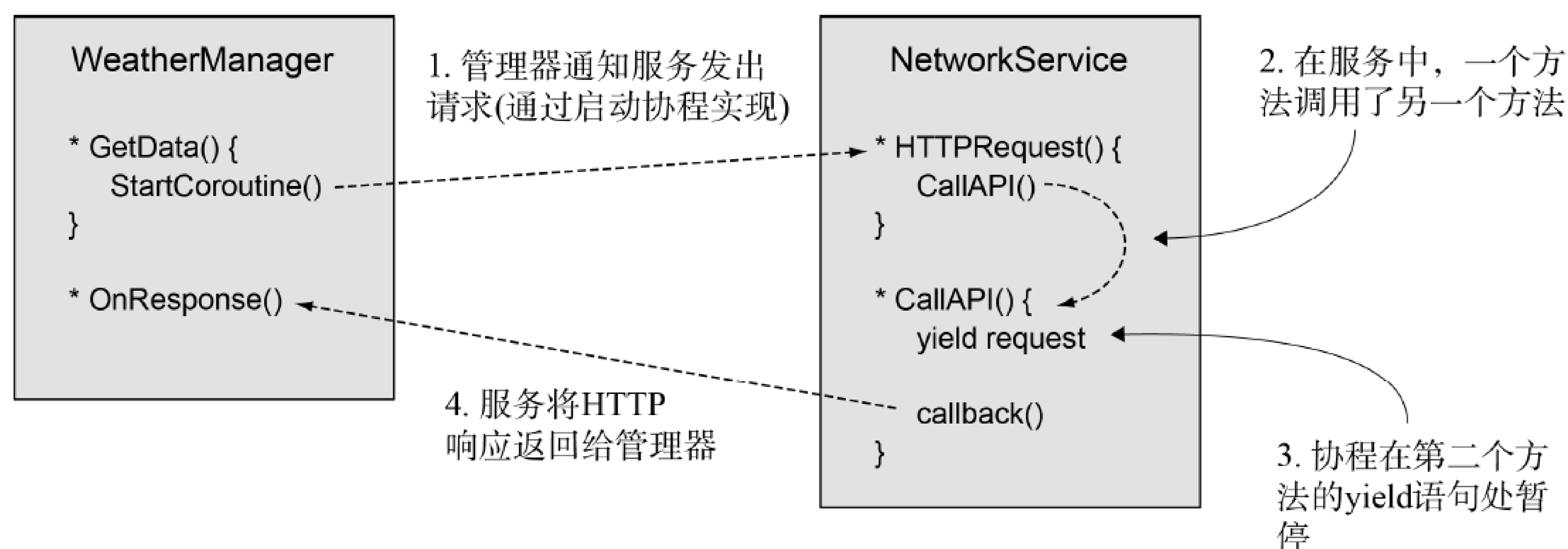


图 10-4　展示网络协程工作原理的图

下一个可能令人不解的是 Action 类型的 callback 参数。

了解回调的工作原理

当协程启动时，会使用 callback 参数来调用方法，callback 参数的类型为 Action。但什么是 Action 呢？

定义　Action 类型是委托(C#有一些委托方法，但这种方法是最简单的)。委托是对其他一些方法/函数的引用。它们允许将函数(或者函数指针)存储在变量中，并把该函数作为参数传给其他函数。

委托允许在函数中传递，就像传递数字和字符串一样。没有委托就无法传递用于后续调用的函数——只能立即调用函数。有了委托，就可以让代码在后续调用其他方法。这在很多情况下很有用，特别是用于实现回调函数。

定义　callback 是用于和调用对象通信的方法。对象 A 可以向对象 B 通知关于 A 中的一个方法。对象 B 可以在之后调用 A 的方法，与 A 通信。

例如，在这个例子中，回调用于将 HTTP 请求完成后返回的数据传回。在 CallAPI()方法中，代码首先创建 HTTP 请求，接着执行 yield 语句直到请求完成，最后使用 callback()发回返回的数据。

注意，Action 关键字使用<>语法。尖括号中的类型声明了这个 Action 需要的参数。换句话说，这个 Action 指向的函数必须带有和尖括号中所声明类型匹配的参数。在这个例子中，参数是一个字符串，因此回调方法的签名必须如下所示：

```
MethodName(string value)
```

在运行了代码清单 10.6 后，就会更清楚地了解回调的概念，这是对回调概念的初次介绍，读者看到更多代码时，就会知道它的实际作用。

代码清单 10.5 中剩余的代码比较直接明了。请求对象是在 using 语句中创建的，

这样一旦完成，对象的内存就会被清理干净。IsResponseValid()条件检查 HTTP 响应中的错误。有两种类型的错误：由于网络连接错误导致的失败，或者由于某些情况导致返回的数据出错。代码中还定义了一个用于产生请求的 URL 常量值(应使用自己的 OpenWeatherMap API 键替换末尾的<your api key>)。

使用联网代码

上面已经在 NetworkService 中封装好了代码。接下来在 WeatherManager 中使用 NetworkService，代码清单 10.6 展示了脚本中增加的代码。

代码清单 10.6 调整 WeatherManager，以使用 NetworkService

```
...
public void Startup(NetworkService service) {
    Debug.Log("Weather manager starting...");

   _network = service;
    StartCoroutine(_network.GetWeatherXML(OnXMLDataLoaded));   ← 开始从互联网加载数据

   status = ManagerStatus.Initializing;   ← 将状态修改为 Initializing 而不是 Started
}

public void OnXMLDataLoaded(string data) {   ← 一旦数据被加载，则回调方法
    Debug.Log(data);

    status = ManagerStatus.Started;
}
...
```

对这个管理器中的代码做了三个主要的修改：启动协程以从互联网下载数据，设置不同的启动状态，定义回调方法以接收响应。

启动协程很简单。在协程背后的大多数复杂操作已经在 NetworkService 中处理了，因此只需要调用 StartCoroutine()。接着设置不同的启动状态，因为管理器还没有完成初始化，它需要在启动完成前从互联网接收数据。

警告 通常使用 StartCoroutine()启动联网方法，一般不直接调用方法。这很容易忘记，因为在协程外创建请求对象不会产生任何编译错误。

调用 StartCoroutine()方法时，你需要调用方法。也就是说，要真正输入圆括号()而不只是函数名。在本例中，协程方法需要一个回调函数作为它的参数，因此要定义那个函数。接下来使用 OnXMLDataLoaded()作为回调函数。注意，这个方法有一个字符串参数，这和 NetworkService 中声明的 Action<string>相符。回调函数现在还没有做很多工作，Debug 那一行简单地将接收的数据打印到控制台，以验证数据是否接收正确。接着 OnXMLDataLoaded 函数的最后一行改变了管理器的启动状态，通知管理器已经完成启动。

单击 Play 运行代码。假设有稳定的网络连接，就会在控制台中显示一串数据，该数据是一个简单的长字符串，但这个字符串以可用的特殊方式格式化。

10.2.2　解析 XML

数据以长字符串的形式存在，通常有少量信息嵌在那个字符串中。可以通过解析字符串提取那些信息。

定义 *解析意味着分析一串数据，把它们分解为独立的信息块。*

为了解析字符串，需要以一种方式格式化数据，允许用户(或解析代码)识别独立的数据块。在互联网上传输数据有一些标准的常用格式，最常用的标准格式为 XML。

定义 *XML 是 Extensible Markup Language(可扩展标记语言)的缩写。它是以结构化方式编码文档的一系列规则，类似 HTML 网页。*

幸运的是，Unity(或 Mono，Unity 内置的代码框架)提供了解析 XML 的功能。我们请求的天气数据格式化为 XML，因此将添加代码到 WeatherManager 中，以解析返回的数据并提取多云信息。将 URL 输入到 Web 浏览器中，观察返回的代码。返回的代码很多，但我们只关心包含类似<clouds value="40"name="scattered clouds"/>的节点。

除了添加代码解析 XML 外，还将使用消息系统(messenger system)。因为下载和解析天气数据后，仍然需要通知场景。创建称为 Messenger 的脚本并将 Unify 维基页面 http:// wiki.unity3d.com/index.php/CSharpMessenger_Extended 中的代码粘贴到脚本中。

接着需要创建一个称为 GameEvent(如代码清单 10.7 所示)的脚本。这个消息系统为剩余代码的事件通信提供了一种解耦方式。

代码清单 10.7　GameEvent 代码

```
public static class GameEvent {
   public const string WEATHER_UPDATED = "WEATHER_UPDATED";
}
```

一旦准备好了消息系统，就按照代码清单 10.8 所示调整 WeatherManager。

代码清单 10.8　在 WeatherManager 中解析 XML

```
...
using System;
using System.Xml;                        ← 确保添加了需要的 using 语句
...
public float cloudValue {get; private set;}← 多云值对外只读，内部可以修改
...
```

```
public void OnXMLDataLoaded(string data) {
   XmlDocument doc = new XmlDocument();       将 XML 解析为一个
   doc.LoadXml(data);                          可搜索的结构
   XmlNode root = doc.DocumentElement;
                                               从数据中拉取
   XmlNode node = root.SelectSingleNode("clouds");   一个节点
   string value = node.Attributes["value"].Value;
   cloudValue = Convert.ToInt32(value) / 100f;   将值转换为 0-1
   Debug.Log("Value: " + cloudValue);            的浮点数

  Messenger.Broadcast(GameEvent.WEATHER_UPDATED);
                                               广播消息，通知
   status = ManagerStatus.Started;             其他脚本
}
...
```

可以看到，最重要的改变在 OnXMLDataLoaded()中。之前这个方法简单地将数据记录到控制台，以验证数据被正确接收。这个代码清单添加了很多用于解析 XML 的代码。

首先创建一个新的空 XML 文档，该文档是一个空容器，可以使用所解析的 XML 结构填充它。下一行代码将数据字符串解析为 XML 文档中包含的结构。接着从 XML 树的根节点开始搜索，以便后续的代码可以搜索到整个 XML 树，找到所有数据。

此时可以在 XML 结构中搜索节点，以便获取需要的信息。在本例中<clouds>是我们唯一感兴趣的节点。首先在 XML 文档中找到该节点，接着从该节点中提取 value 属性。该属性数据定义了 cloud 的值为 0～100 的整型值，但我们需要把它调整为 0～1 的浮点数，以便于后面调整场景。进行这一步转换只需要向代码中添加一个简单的数学函数。

最后，在从完整的数据中提取出多云的值之后，广播一个天气数据已经更新的消息。当前代码没有侦听这个消息的侦听器，但广播者不需要了解任何与侦听器相关的信息(实际上，这是解耦消息系统的要点)。随后将侦听器添加到场景中。

前面编写了解析 XML 数据的代码！但在将这些值应用到可视场景之前，先介绍另一种数据传输的方法。

10.2.3　解析 JSON

在继续该项目的下一步之前，先探讨另一种传输数据的格式。XML 是一种通用的数据传输格式，另一种通用格式是 JSON。

定义　JSON 是 JavaScript Object Notation 的缩写。与 XML 的目标类似，JSON 被设计为轻量级的格式。尽管 JSON 的语法源于 JavaScript，但这种格式是没有限定语言的，实际上它可用于不同种类的编程语言。

不像 XML，Mono 没有包含这种格式的解析器。可以从网上下载一些优秀的 JSON 解析器，例如 MiniJSON(https://gist.github.com/darktable/1411710)和 SimpleJSON (http://wiki.unity3d.

com/index.php/SimpleJSON)。

这个例子使用 MiniJSON。创建一个脚本，命名为 MiniJSON，并将上面网址中的代码粘贴到脚本中。现在可以使用这个库来解析 JSON 数据。我们已经从 OpenWeatherMap API 获取 XML，但由于可以将相同的数据以 JSON 格式发送，因此根据代码清单 10.9 中的代码修改 NetworkService。

代码清单 10.9　使 NetworkService 请求 JSON 而不是请求 XML

```
...
private const string jsonApi =                           ← 此处的 URL
"http://api.openweathermap.org/data/2.5/weather?q=Chicago,us&APPID=<your api   稍微有点不同
  key>";
...
public IEnumerator GetWeatherJSON(Action<string> callback) {
   return CallAPI(jsonApi, callback);
}
...
```

上面的代码类似于下载 XML 数据的代码，只是 URL 有点不同。这个请求返回的数据与请求 XML 返回的值一样，但格式不同。这次需要查找类似"clouds":{" all":40}的块。

这次不需要一大堆额外的代码，因为我们已将请求代码封装到独立的函数中，这样以后可以很轻松地将每个 HTTP 请求添加到游戏中。很好！现在修改 WeatherManager，请求 JSON 数据而不是请求 XML(如代码清单 10.10 所示)。

代码清单 10.10　修改 WeatherManager，请求 JSON

```
...                                                  确保添加所需
using MiniJSON;                                   ←  的 using 语句
...
public void Startup(NetworkService service) {
  Debug.Log("Weather manager starting...");

     _network = service;
  StartCoroutine(_network.GetWeatherJSON(OnJSONDataLoaded));   ← 改变的网络请求

   status = ManagerStatus.Initializing;
}
...
public void OnJSONDataLoaded(string data) {          不使用自定义的 XML 容器，而
   Dictionary<string, object> dict;               ←  是解析到字典中
   dict = Json.Deserialize(data) as Dictionary<string,object>;

    Dictionary<string, object> clouds = (Dictionary<string,object>)


      dict["clouds"];
    cloudValue = (long)clouds["all"] / 100f;         语法已经改变，
    Debug.Log("Value: " + cloudValue);               但这些代码的
                                                     作用依然相同
```

```
        Messenger.Broadcast(GameEvent.WEATHER_UPDATED);

        status = ManagerStatus.Started;
    }
    ...
```

可以看出，使用 JSON 的代码看起来和使用 XML 的代码很类似。唯一的区别是 JSON 解析器使用标准的 Dictionary，而 XML 解析器使用自定义文档容器。有一个命令用于反序列化(deserialize)，而反序列化这个词可能大家还不熟悉。

定义 *反序列化意味着和解析一样的处理。这是序列化的相反操作，意味着将一批数据编码为一种能传输和存储的格式，例如 JSON 字符串。*

除了语法不同，所有步骤都一样。从数据块中提取值(出于某些原因，此时的值称为 all，但那只是 API 的习惯)，通过简单的数学知识将值转换为 0-1 的浮点数。

完成上述操作后，现在将值应用到可见场景中。

10.2.4　基于天气数据影响场景

不管数据具体是如何格式化的，一旦从响应数据中提取出多云值，就可以在 WeatherController 的 SetOvercast()方法中使用该值。不管使用 XML 或 JSON，数据字符串最后都解析为一系列单词和数字。SetOvercast()方法使用数字作为参数。在 9.1.2 节使用的是每帧递增的数字，也可以很方便地使用由天气 API 返回的数字。

代码清单 10.11 展示了修改后的 WeatherController 脚本的完整代码。

代码清单 10.11　对所下载的天气数据进行响应的 WeatherController

```
using UnityEngine;
using System.Collections;

public class WeatherController : MonoBehaviour {
    [SerializeField] private Material sky;
    [SerializeField] private Light sun;

    private float _fullIntensity;                    // 添加/移除事件侦听器

    void Awake() {
        Messenger.AddListener(GameEvent.WEATHER_UPDATED, OnWeatherUpdated);
    }
    void OnDestroy() {
        Messenger.RemoveListener(GameEvent.WEATHER_UPDATED, OnWeatherUpdated);
    }

    void Start() {
        _fullIntensity = sun.intensity;
    }
```

```
    private void OnWeatherUpdated() {
      SetOvercast(Managers.Weather.cloudValue);
    }

   private void SetOvercast(float value) {
      sky.SetFloat("_Blend", value);
      sun.intensity = _fullIntensity - (_fullIntensity * value);
    }
}
```

使用 WeatherManager 的多云值

注意，此处的修改不仅添加了一些代码，还移除了一些测试代码。具体而言，移除了由每帧递增的本地多云值，现在已不再需要该值了，因为后面将使用从 WeatherManager 中返回的值。

在 Awake()中添加了一个侦听器，在 OnDestroy()中移除它(Awake()和 OnDestroy()是唤醒或移除对象时调用的两个 MonoBehaviour 函数)。侦听器是广播消息系统的一部分，当收到消息时调用 OnWeatherUpdated()。OnWeatherUpdated()从 WeatherManager 中获取多云值，并使用该值调用 SetOvercast()。通过这种方式，场景的外观由下载的天气数据控制。

现在运行场景，天空会根据天气数据的多云值而变化。请求天气数据花费了一些时间。在真正的游戏中，可以将场景隐藏在一个加载屏幕后，直到天空更新完成。

HTTP 以外的游戏网络

HTTP 请求是健壮、可靠的，但对于大多数游戏而言，发送请求和接收响应之间的延迟可能有些长。因此 HTTP 请求是发送速度相对较慢的消息到服务器的一种好方式(例如，在基于回合的游戏中移动或为任何游戏提交高分请求)，但类似多人 FPS 的游戏则需要另一种联网方式。

该方式包括不同的通信技术，也包括延迟补偿技术。例如，Unity 为多人游戏提供了一个高级 API，它建立在低级传输层的基础之上。

联网动作游戏的前沿是一个复杂的主题，它超出了本书的讨论范围。更多的相关信息可以访问 http://docs.unity3d.com/Manual/NetworkReferenceGuide.html。

现在知道了如何从互联网获取数字和字符串数据，接下来对图像做相同的处理。

10.3　添加一个网络布告栏

从 Web API 返回的通常是 XML 或 JSON 格式的文本字符串，还有很多其他类型的数据通过互联网传输，最常见的被请求的数据类型是图像。UnityWebRequest 对象也可以用于下载图像。

要完成这个任务，需要创建一个布告栏，显示从互联网下载的图像。需要分两个步骤进行编码：下载用于显示的图像，将图像应用到布告栏对象。第三步是改善代码，

保存图像，以用于多个布告板。

10.3.1　从互联网加载图像

首先，编写代码来下载图像。接下来下载一些用于测试的公共领域的风景图像(如图 10-5 所示)。下载的图像还不会显示在布告栏上，下一节会展示显示图像的脚本，但现在先准备好获取图像的代码。

下载图像和下载数据的代码架构看起来很类似。我们使用一个新的管理器模块(称为 ImagesManager)来控制要显示的下载图像。同样，连接到互联网并发送 HTTP 请求的细节由 NetworkService 处理，而 ImagesManager 将调用 NetworkService 来下载所需的图像。



图 10-5　加拿大班夫国家公园的梦莲湖图像

添加的第一处代码在 NetworkService 中。代码清单 10.12 将下载图像的代码添加到脚本中。

代码清单 10.12　在 NetworkService 中下载图像

```
...
private const string webImage =                                  将这个常量和其
"http://upload.wikimedia.org/wikipedia/commons/c/c5/             他 URL 放在顶部
    Moraine_Lake_17092005.jpg";
...                                         这个回调使用 Texture2D 而不是使用字符串
public IEnumerator DownloadImage(Action<Texture2D> callback) {
  UnityWebRequest request = UnityWebRequestTexture.GetTexture(webImage);

  yield return request.Send();
  callback(DownloadHandlerTexture.GetContent(request));
                                             使用 DownloadHandler 工具
}                                            获得下载的图像
...
```

下载图像的代码看起来和下载数据的代码几乎相同。主要区别在于回调方法的类型。注意，回调方法这次使用的是 Texture2D 参数，而不是使用字符串。这是因为我

们发送回了对应的响应：之前下载的是字符串数据，而现在下载的是图像。代码清单 10.13 包含新 ImagesManager 的代码。创建一个新脚本，并输入那些代码。

代码清单 10.13　创建 ImagesManager，用于获取并存储图像

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System;

public class ImagesManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  private NetworkService _network;

  private Texture2D _webImage;   ← 用于存储所下载图像的变量

  public void Startup(NetworkService service) {
    Debug.Log("Images manager starting...");

    _network = service;

    status = ManagerStatus.Started;
  }

  public void GetWebImage(Action<Texture2D> callback) {
    if (_webImage == null) {                              ← 检查图像是否已经存储
      StartCoroutine(_network.DownloadImage(callback));
    }
    else {
      callback(_webImage);   ← 如果图像已经存储，立刻调用(不是下载)回调
    }
  }
}
```

这段代码最有趣的部分是 GetWebImage()，该脚本中的其他部分由标准属性和实现管理器界面的方法组成。当调用 GetWebImage()时，它返回(通过回调函数)Web 图像。首先它将检查_webImage 是否有存储的图像。如果没有，就进行网络调用，下载图像，否则 GetWebImage()将返回存储的图像(而不是重新下载图像)。

注意 当前，下载的图像从未存储，这意味着_webImage 将一直为空。代码指定了当_webImage 不为空时该如何处理，因此接下来的部分将调整代码，存储图像。调整代码之所以单独放在一节中讲解，是因为它包含了一些代码技巧。

当然，就像所有管理器模块一样，需要将 ImagesManager 添加到 Managers 中。代码清单 10.14 展示了将 ImagesManager 添加到 Managers.cs 中的细节。

代码清单 10.14　将新的管理器添加到 Managers.cs 中

```
...
[RequireComponent(typeof(ImagesManager))]
...
public static ImagesManager Images {get; private set;}
...
void Awake() {
   Weather = GetComponent<WeatherManager>();
   Images = GetComponent<ImagesManager>();

   _startSequence = new List<IGameManager>();
   _startSequence.Add(Weather);
   _startSequence.Add(Images);

   StartCoroutine(StartupManagers());
}
...
```

与 WeatherManager 的设置不同，ImagesManager 中的 GetWebImage()在启动时不会自动调用，而是会等待直到被调用。下一节中将对此进行介绍。

10.3.2　在布告栏上显示图像

刚刚编写的 ImagesManager 在调用前不会执行任何操作，因此现在创建一个布告栏对象，并调用 ImagesManager 中的方法。首先创建一个新的立方体，把它放在场景的中间，例如，位置为(0, 1.5, -5)，大小为(5, 3, 0.5)(如图 10-6 所示)。

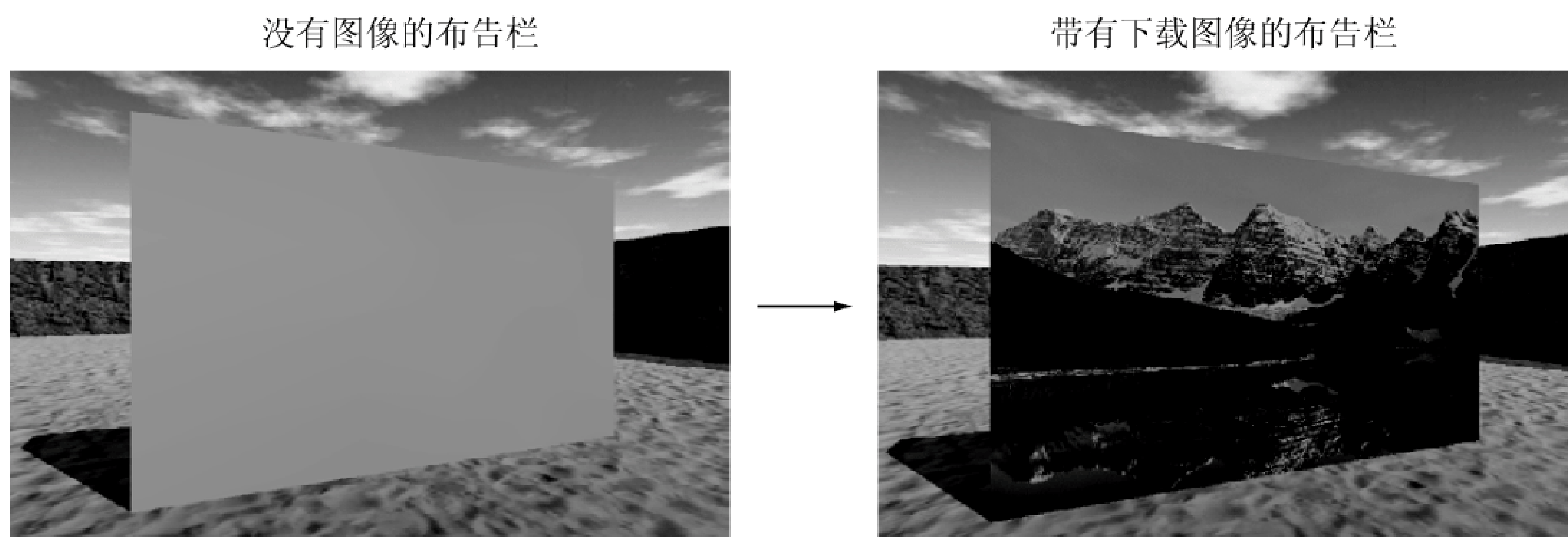


图 10-6　显示下载图像前后的布告栏对象

接下来将创建类似第 9 章中变色显示器的设备。复制 DeviceOperator 脚本，将它放到玩家上。如前所述，在按下 Fire3 按键(在项目的输入设置中，该键定义为左 Shift 键)时，脚本将操作附近的设备。另外，为布告栏设备创建一个名为 WebLoadingBillboard 的脚本，将该脚本添加到布告栏对象上，并输入代码清单 10.15 中的代码。

代码清单 10.15　WebLoadingBillboard 设备脚本

```
using UnityEngine;
using System.Collections;

public class WebLoadingBillboard : MonoBehaviour {
  public void Operate() {
    Managers.Images.GetWebImage(OnWebImage);    ← 调用 ImagesManager 中的方法
  }

  private void OnWebImage(Texture2D image) {
    GetComponent<Renderer>().material.mainTexture = image;    ← 在回调中将已经下载的图像应用到材质
  }
}
```

这段代码完成了两个主要操作：当设备运行时调用 ImagesManager.GetWebImage()，从回调函数中应用图像。由于贴图应用到材质上，因此可以修改布告栏材质的贴图。图 10-6 展示了在游戏运行后显示的布告栏。

AssetBundles：如何下载其他类型的资源

使用 UnityWebRequest 下载图像相当简单，但如何下载其他类型的资源(例如，网格对象和预设)呢？UnityWebRequest 有用于文本和图像的属性，但使用它下载其他资源会比较复杂。

Unity 可以通过 AssetBundles 机制下载任何类型的资源。简言之，就是首先将一些资源打包，接着 Unity 就可以在下载该包后解压缩资源。创建和下载 AssetBundle 超出了本书的讨论范围。如果想学习更多相关知识，可以从阅读 Unity 手册的这部分内容开始入手：http://docs.unity3d.com/Manual/AssetBundlesIntro.html 和 https://docs.unity3d.com/Manual/UnityWebRequest-DownloadingAssetBundle.html。

下载的图像已经显示在布告栏上！可以对这段代码进一步优化，以处理多个布告栏。在下节中将进行这个优化。

10.3.3　缓存下载的图像以供重用

如 9.3.1 节中所述，ImagesManager 还没有保存所下载的图像。这意味着该图像重复下载才能用于多个布告栏。这比较低效，因为每次显示的都是相同的图像。为了优化这一点，接下来调整 ImagesManager，缓存已下载的图像。

定义　缓存意味着在本地保存。最常用(但不是唯一)的情形是从互联网下载的图像。

关键是要在 ImagesManager 中提供一个回调函数，先保存图像，接着从 WebLoadingBillboard 调用该回调函数。这有些棘手(与当前代码使用 WebLoadingBillboard 的回调相反)，因为代码事先不知道 WebLoadingBillboard 的回调是什么。换言之，无法在

ImagesManager 中编写一个方法，使之调用 WebLoadingBillboard 中的特定方法，因为代码不知道要调用的具体是哪个方法。解决这个难题的做法是使用 lambda 函数。

定义　lambda 函数(也称为匿名函数)是指没有名称的函数。这种函数通常在其他函数中临时创建。

lambda 函数是一个棘手的代码特性，很多编程语言都支持它，包括 C#。通过为 ImagesManager 中的回调使用 lambda 函数，代码可以临时创建回调函数，使用从 WebLoadingBillboard 中传入的方法。不需要提前知道调用的是什么方法，因为这个 lambda 函数事先并不存在！代码清单 10.16 展示了如何在 ImagesManager 中实现这个技巧。

代码清单 10.16　ImagesManager 中用于回调的 lambda 方法

```
...
using System;
...
public void GetWebImage(Action<Texture2D> callback) {
   if (_webImage == null) {
   StartCoroutine(_network.DownloadImage((Texture2D image) => {
       _webImage = image;
       callback(_webImage);
      }));
   }
   else {
      callback(_webImage);
   }
}
...
```

存储已下载的图像

回调函数在 lambda 函数中使用，而不是直接发送到 NetworkService

主要的修改是传给 NetworkService.DownloadImage()的函数。之前代码传入的是和 WebLoadingBillboard 方法中一样的回调函数。修改后，发送到 NetworkService 的回调函数是声明为一个单独的 lambda 函数，它调用了 WebLoadingBillboard 中的方法。注意，声明 lambda 方法的语法为：()=>{}。

使回调成为一个单独的函数，这样它执行的任务就可以比调用 WebLoadingBillboard 中的方法更多。具体而言，lambda 函数也保存了已下载图像的本地副本。因此 GetWebImage()只是在首次下载图像，所有后续调用将使用本地保存的图像。

因为这个优化是针对后续调用的，所以效果只能在多个布告栏观察到。接下来复制布告栏对象，以便在场景中拥有第二块布告栏。选择布告栏对象，单击 Duplicate(在 Edit 菜单下或者右击)，并移开所复制的布告栏(例如，将 X 位置修改为 18)。

现在运行游戏，观察发生了什么。当操作第一个布告栏时，会注意到当图像从互联网下载时，游戏有显著的停顿。接着移动到第二块布告栏时，图像将立刻出现，因为它已经下载过了。

这是对于下载图像的一个重要优化(这正是 Web 浏览器默认缓存图像的原因)。还

有一个更主要的网络任务需要了解：将数据发送到服务器。

10.4　将数据发送到 Web 服务器

前面介绍了多个下载数据的示例，还需要编写一个发送数据的示例。最后一节要求配备一个用于发送请求的服务器，因此该节的内容是可选的。但下载开源软件并设置用于测试的服务器是十分容易的。

推荐将 XAMPP 用作测试服务器。可以从 www.apachefriends.org 下载 XAMPP。一旦安装完，运行服务器，则可以像访问互联网的服务器一样，通过 http://localhost/ 访问 XAMPP 的 htdocs 文件夹。设置好 XAMPP 并成功运行后，就在 htdocs 中创建称为 uia 的文件夹，用于放置服务器端脚本。

不管是使用 XAMPP 还是使用已有的 Web 服务器，本节的任务都是当玩家到达场景中的检查点时，将天气数据发送到服务器。这个检查点是一个触发空间，类似于第 9 章的门触发器。需要创建一个新的立方体对象，将该对象定位于场景的另一边，将碰撞器设置为 Trigger，并如前面章节所示为该对象应用半透明材质(记住，要设置材质的 Rendering Mode)。图 10-7 展示了一个应用了绿色半透明材质的检查点对象。

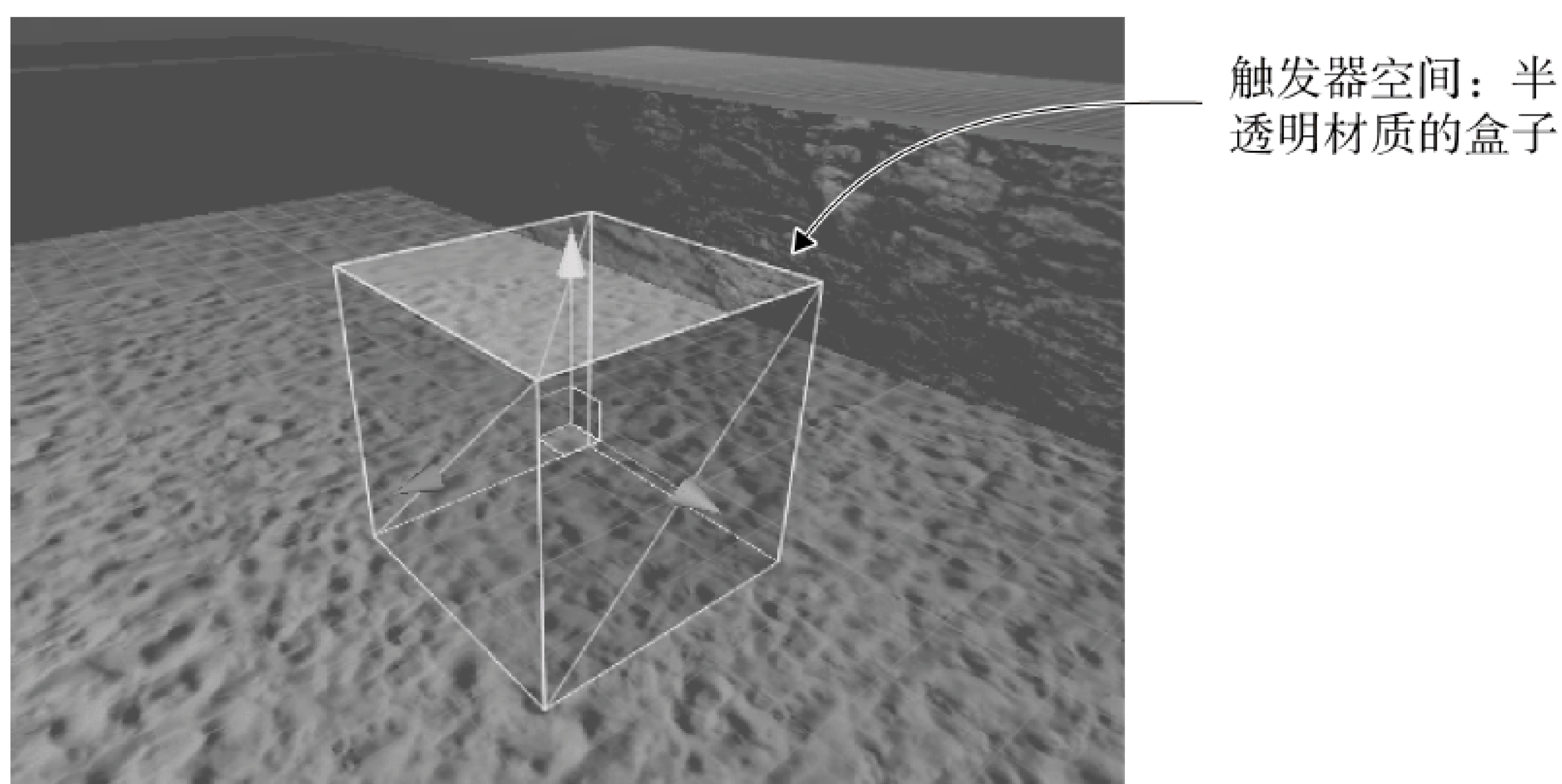


图 10-7　触发数据传输的检查点对象

现在触发器对象已经在场景中，接下来编写它调用的代码。

10.4.1　跟踪当前的天气：发送 post 请求

检查点对象调用的代码将嵌套很多脚本。和下载数据的代码一样，发送数据的代码将包含通知 NetworkService 创建请求的 WeatherManager，和处理 HTTP 通信细节的 NetworkService。代码清单 10.17 展示了需要对 NetworkService 所做的调整。

代码清单 10.17　调整 NetworkService 以发送数据

```
...                                          服务器端脚本的地址，可以根据需要进行修改
private const string localApi = "http://localhost/ch9/api.php"; ←
...
private IEnumerator CallAPI(string url, WWWForm form, Action<string>
    callback) {                              ← 给 CallAPI()添加参数
  using (UnityWebRequest request = (form == null) ?
    UnityWebRequest.Get(url) : UnityWebRequest.Post(url, form)) { ←

    yield return request.Send();             使用 WWWForm 执行 POST，或
                                             者直接执行 GET
    if (request.isError) {
      Debug.LogError("network problem: " + request.error);
    } else if (request.responseCode != (long)System.Net.HttpStatusCode.OK)
   {
      Debug.LogError("response error: " + request.responseCode);
   } else {
      callback(request.downloadHandler.text);
   }
  }
}                                            由于修改了形参，因此也修改调用

public IEnumerator GetWeatherXML(Action<string> callback) {
  return CallAPI(xmlApi, null, callback); ←
}
public IEnumerator GetWeatherJSON(Action<string> callback) {
  return CallAPI(jsonApi, null, callback);
}

public IEnumerator LogWeather(string name, float cloudValue, Action<string>
callback) {
WWWForm form = new WWWForm();               ← 定义了一个带有要发送值的表单
form.AddField("message", name);
form.AddField("cloud_value", cloudValue.ToString());
form.AddField("timestamp", DateTime.UtcNow.Ticks.ToString()); ←
                                             和多云值一起
return CallAPI(localApi, form, callback);    发送时间戳
}
...
```

首先，注意 CallAPI()有一个新形参。这是一个 WWWForm 对象，是与 HTTP 请求一起发送的一系列值。代码中有一个条件，使用 WWWForm 对象的存在来更改创建的请求。通常我们希望发送 GET 请求，但是 WWWForm 把它更改为 POST 请求，以发送数据。代码中的其他变更都响应了这个主要修改(例如，修改 GetWeather 代码，因为 CallAPI()形参变了)。

代码清单 10.18 展示了需要在 WeatherManager 中添加的内容。

代码清单 10.18　将发送数据的代码添加到 WeatherManager 中

```
...
public void LogWeather(string name) {
```

```
        StartCoroutine(_network.LogWeather(name, cloudValue, OnLogged));
    }
    private void OnLogged(string response) {
        Debug.Log(response);
    }
    ...
```

最后，将检查点脚本添加给场景中的触发器空间来使用那段代码。创建一个名为 CheckpointTrigger 的脚本，将脚本添加到触发器空间上，并输入代码清单 10.19 中的内容。

代码清单 10.19 用于触发器空间的 CheckpointTrigger 脚本

```
using UnityEngine;
using System.Collections;

public class CheckpointTrigger : MonoBehaviour {
    public string identifier;

    private bool _triggered;          ←— 如果检查点已被触发，则跟踪它

    void OnTriggerEnter(Collider other) {
        if (_triggered) {return;}

        Managers.Weather.LogWeather(identifier);     ←— 调用以发送数据
        _triggered = true;
    }
}
```

在 Inspector 中会出现一个标识符槽，将它命名为 checkpoint1。运行代码，进入检查点时，数据会发送出去。不过，响应指示出现了一个错误，这是因为服务器还没有接收请求的脚本。本节的最后部分将编写该脚本。

10.4.2 PHP 中的服务器端代码

服务器端需要有脚本来接收从游戏发送的数据。编写服务器端脚本超出了本书的讨论范围，因此这里不详细描述。在此仅快速编写一个 PHP 脚本，因为这是最容易的方式。在 htdocs 中(或者存放 Web 服务器的位置)创建一个文本文件，并命名为 api.php(见代码清单 10.20)。

代码清单 10.20 使用 PHP 编写的接收数据的服务器脚本

```
<?php

$message = $_POST['message'];          ←— 将 post 数据解压缩到变量中
$cloudiness = $_POST['cloud_value'];
 $timestamp = $_POST['timestamp'];
 $combined = $message."cloudiness=".$cloudiness."time=".$timestamp."\n";
```

```
$filename = "data.txt";                          ← 定义要写入的文件名
file_put_contents($filename, $combined, FILE_APPEND | LOCK_EX);  ← 写入文件

echo "Logged";

?>
```

注意，这个脚本将接收到的数据写入 data.txt，因此也需要在服务器上放置一个 data.txt 文本文件。一旦 api.php 准备就绪，当触发游戏中的检查点时，天气日志就出现在 data.txt 文件中。

10.5　小结

- 天空盒用于在所有对象背后渲染天空视觉效果
- Unity 提供了 UnityWebRequest，用于下载数据
- XML 和 JSON 等通用数据格式很容易解析
- 材质可以显示从互联网下载的图像
- UnityWebRequest 可以将数据发送到 Web 服务器

第 11 章

播放音频：音效和音乐

本章涵盖：

- 为不同音效导入并播放音频剪辑
- 将 2D 音效用于 UI，3D 音效用于场景
- 当播放音效时调整所有音效的音量
- 运行游戏时播放背景音乐
- 在不同的背景曲调之间淡入淡出

在视频游戏中，尽管图形作为其内容得到了最多的关注，但音频也很重要。大多数游戏都播放背景音乐和音效。因此 Unity 也提供了音频功能，以便在游戏中播放背景音乐和音效。Unity 可以导入和播放各种不同的音频文件格式，调整音量，甚至处理场景中特定位置的音效。

注意 对于 2D 和 3D 游戏，音频处理方式都是相同的。尽管本章中的示例项目是一个 3D 游戏，但本章的所有内容都适用于 2D 游戏。

本章从音效而不是音乐开始介绍。音效是比较短的声音剪辑，它在游戏中随着动作播放(例如当玩家开火时播放枪击声)。然而对于音乐，声音剪辑比较长(通常运行几分钟)，且在游戏中回放不直接绑定事件。最终，两者虽然都是音频文件，回放代码也相同，但音乐声音文件通常比音效声音文件更大(实际上，音乐文件通常是游戏中最大的文件！)。

本章完整的路线图将会从一个没有声音的游戏开始，完成如下工作：

(1) 导入音效的音频文件

(2) 为敌人和射击播放音效

(3) 编写一个音频管理器控制音量

(4) 优化音乐的加载

(5) 分别控制音乐和音效的音量，包括淡入淡出轨道

注意 本章很大程度上独立于前面构建的项目，只是简单地将上述的音乐特性添加到已有的游戏示例中。本章的所有示例均构建在第 3 章创建的 FPS 之上，可以下载该示例项目，也可以使用自己喜欢的任何游戏示例。

将已有的游戏示例复制到本章后，就可以开始处理第一步了：导入音效。

11.1　导入音效

在播放任何音效之前，很明显，需要将音效文件导入到 Unity 项目中。首先以所需的文件格式收集音效剪辑，接着将文件带到 Unity 中并根据需要调整它们。

11.1.1　支持的文件格式

与第 4 章的美术资源相同，Unity 支持不同类型的各具优缺点的音频格式。表 11-1 列出 Unity 支持的音频格式。

表 11-1　Unity 支持的音频文件格式

文 件 类 型	优　缺　点
WAV	Windows 上默认的音频格式。未压缩的声音文件
AIF	Mac 上默认的音频格式。未压缩的声音文件
MP3	压缩的声音文件；文件更小，但以牺牲一点质量为代价
OGG	压缩的声音文件；文件更小，但以牺牲一点质量为代价
MOD	音轨文件格式。专业类型的高效数字音乐
XM	音轨文件格式。专业类型的高效数字音乐

不同音频文件最主要考虑的因素是它们所应用的压缩方式。压缩操作减小了文件的大小，但是会丢失一些文件信息。音频压缩很聪明地丢弃了最不重要的信息，以便压缩后的声音听起来还不错。然而，它还是会导致微小的质量损耗，因此当声音剪辑比较短而且文件不大时，应该选择未压缩的音频。长声音剪辑(特别是音乐)应该使用已压缩的音频，因为不这样做，音频剪辑将会相当大。

不过，Unity 为决定是否压缩提供了一个小便利。

提示 尽管音乐在最终的游戏中应压缩，但 Unity 可以在导入文件之后压缩音频。因此，在 Unity 中开发游戏时，通常可以使用未压缩的文件格式，甚至是长音乐也可以这样做，而不是导入已压缩的音频。

数字音频的工作原理

通常，音频文件保存声音播放时由扬声器创建的波形。声音是一系列的波，它们能通过空气传播，不同的声音，声波的大小和频率不同。音频文件记录这些波时，会反复在短暂的时间间隔进行采样，并保存每次采样波的状态。

采样波记录得越频繁，就越能精确记录波形随时间变化的细节——两次改变之间的间隙也就越小。但更频繁的采样意味着有更多的数据需要保存，所得的文件也就越大。压缩的声音文件通过一系列技巧减小文件的大小，包括丢弃不被听众注意的声音频率。

音轨是一种特殊类型的用于创建音乐的软件音序器。鉴于传统音乐文件保存声音的原始波形，音序器保存一些更类似于乐谱的信息：轨道文件是一系列注释，每个注释带有强度和音高等信息。这些注释组成波形，但减少了保存的数据总量，因为相同的注释在整个序列中重复使用。以这种方式合成的音乐会更高效，但这是一种相当专业的音频。

因为 Unity 在导入音频后会压缩音频，所以通常应该选择 WAV 或者 AIF 文件格式。短音效和长音乐可能需要分别调整导入设置(尤其是，告诉 Unity 什么时候应该应用压缩)，但原始文件通常是不压缩的。

创建声音文件有多种方式(例如，附录 B 提到，Audacity 工具可以记录麦克风中的声音)，但本例从一个免费声音网站中下载一些声音。使用从 www.freesound.org 下载的一些 WAV 剪辑。

警告 “免费”声音在不同的许可方案中提供，因此确保能够以想要的方式使用这些声音剪辑。例如，很多免费声音只能用于非商业用途。

本示例项目使用下面公共的音效(当然，可以选择下载自己的音效，请留心其中列出的许可)：

- “thump”由 hy96 制作
- “ding”由 Daphne_in_Wonderland 制作
- “swish bamboo pole”由 ra_gun 制作
- “fireplace”由 leosalom 制作

一旦有了游戏中需要的声音文件，下一步便是将这些声音导入到 Unity 中。

11.1.2 导入音频文件

收集了一些音频文件后，需要将它们导入到 Unity 中。如第 4 章中对美术资源的处理一样，必须将音频资源导入到 Unity 中，游戏才能使用它们。

导入音频文件机制很简单，和导入其他资源的机制相同：从计算机上文件所在的位置将它们拖动到 Unity 的 Project 视图上(创建 Sound FX 文件夹，并将音频文件拖入其中)。就是这么简单！不过和其他资源一样，音频文件在 Inspector 中也有用于调整的导入设置(如图 11-1)。

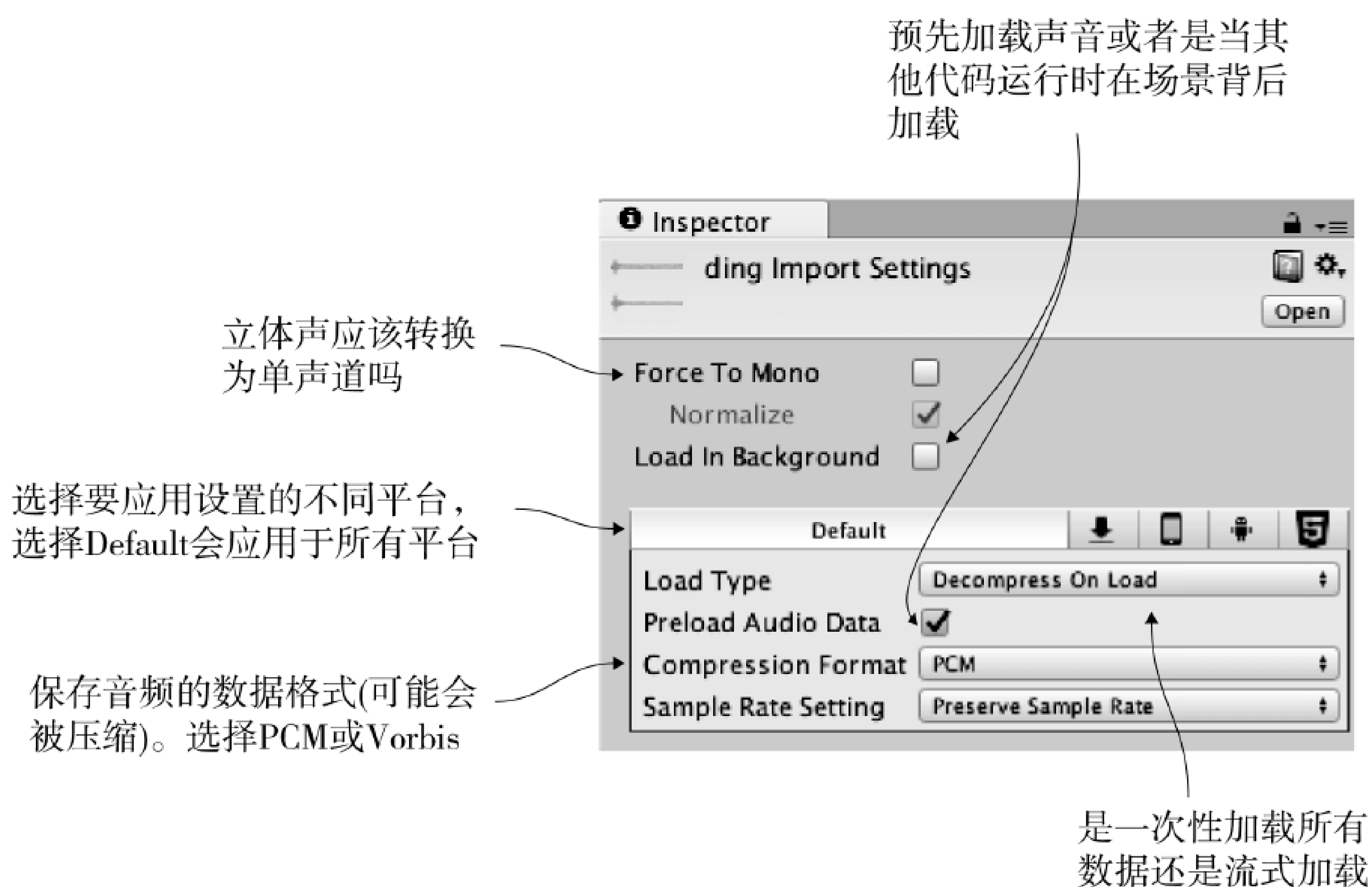


图 11-1 音频文件的导入设置

不要选中 Force To Mono 复选框，该复选框指的是单声道和立体声。通常声音都是以立体声记录的，立体声实际上在文件中记录了两个波形，一个用于左扬声器，一个用于右扬声器。为了减小文件尺寸，可以将音频信息减半，把相同的波形发送到两个扬声器，而不是分别发送到左右扬声器。(还有一个 Normalize 设置，只要打开 Mono 才应用该设置，所以关闭 Mono 时它是灰显的。)

在 Force To Mono 的下方是 Load In Background 和 Preload Audio Data 复选框。预加载设置与回放性能和内存使用相关。等待使用声音时，预加载音频会消耗内存，但可以避免等待加载。因此，不要预加载长音频剪辑，但要为短的声效打开该设置。在程序后台加载音频，允许程序在音频加载时一直运行，这通常适合于长的音乐剪辑，可以使程序不会停滞。但这意味着音频不会立刻开始播放。通常，对于短音频剪辑，应该关闭这个设置，以确保它们能在播放前加载完全。因为导入的剪辑都是短的音频，所以应该关闭 Load In Background。

最后，最重要的设置是 Load Type 和 Compression Format。Compression Format 控

制存储音频数据的格式。如前所述，音乐应该被压缩，本例中选择 Vorbits(这是一种压缩音频格式的名称)。短的声音剪辑不需要压缩，因此为这些剪辑选择 PCM(Pulse Code Modulation，原始、采样的声波的技术术语)。第三个设置 ADPCM 是 PCM 的变体，偶尔能获得稍微好一点的声音质量。

Load Type 控制计算机加载文件中数据的方式。由于计算机的内存有限，而音频文件可能会比较大，因此有时想让音频播放的同时数据流入内存，以避免计算机一次性将整个文件加载到内存中。但这样的处理在流传输音频时会增加一点计算开销，因此当音频首次加载到内存后，音频的播放性能最佳。即使那样，也可以选择音频数据是以压缩格式加载，还是为了快速回放而不压缩。由于这些声音剪辑很短，因此它们不需要流式传输，且可以设置为 Decompress On Load。

最后一个设置是 Sample Rate。保持 Preserve Sample Rate 不变，这样 Unity 就不会改变导入文件中的样本了。此时，所有导入的音效就可以使用了。

11.2　播放音效

在项目中添加了一些声音文件后，接下来自然是播放这些声音。触发音效的代码不会很难理解，但 Unity 中的音频系统包含必须一起正确工作的许多部分。

11.2.1　音频剪辑、音源和声音侦听器

虽然在播放声音时，只需要简单地告诉 Unity 要播放哪个剪辑即可，但为了在 Unity 中播放声音，必须定义三个不同的部分：AudioClip、AudioSource 和 AudioListener。将声音系统分离为多个组件的原因是 Unity 对 3D 声音的支持：不同的组件告诉 Unity 位置信息，以便它管理 3D 声音。

2D 和 3D 声音

游戏中的声音可以是 2D 或 3D 的。2D 声音我们已熟悉，即正常播放的标准音频。“2D 声音”通常意味着“不是 3D 声音”。

3D 声音仅限于 3D 模拟，读者可能还不熟悉，3D 声音在模拟中有指定的位置。它们的音量和音高受到侦听器移动的影响。例如，远处触发的音效听起来会很微弱。

Unity 支持各种类型的音频，而由用户决定音源应该是播放 2D 声音还是 3D 声音。类似音乐之类的声音应该是 2D 声音，而在场景中为大多数声音效果使用 3D 声音将会创建一种沉浸感。

作为类比，想象真实世界中的房间。房间有一套立体声系统正在播放 CD。如果有个人走进房间，他听得清楚。当他离开房间时，他听的声音越来越轻，甚至最后听不到。类似的，如果在房间中移动立体声系统，随着系统的移动，音乐声音会发生变

化。如图 11-2 所示，在这个类比中，CD 是 AudioClip，立体声系统是 AudioSource，人是 AudioListener。

在这三个不同的部分中，首先是 AudioClip。AudioClip 是指上一节导入的声音文件。原始波形数据是音频系统处理任何事情的基础，但音频剪辑本身不做任何事情。

下一种对象是 AudioSource。AudioSource 播放声音剪辑。这是音频系统实际作用的抽象，而它是一个有用的抽象，使 3D 声音更易于理解。3D 声音从指定的音源播放，它位于该音源的位置。2D 声音通常必须从音源播放，但与音源的位置无关。

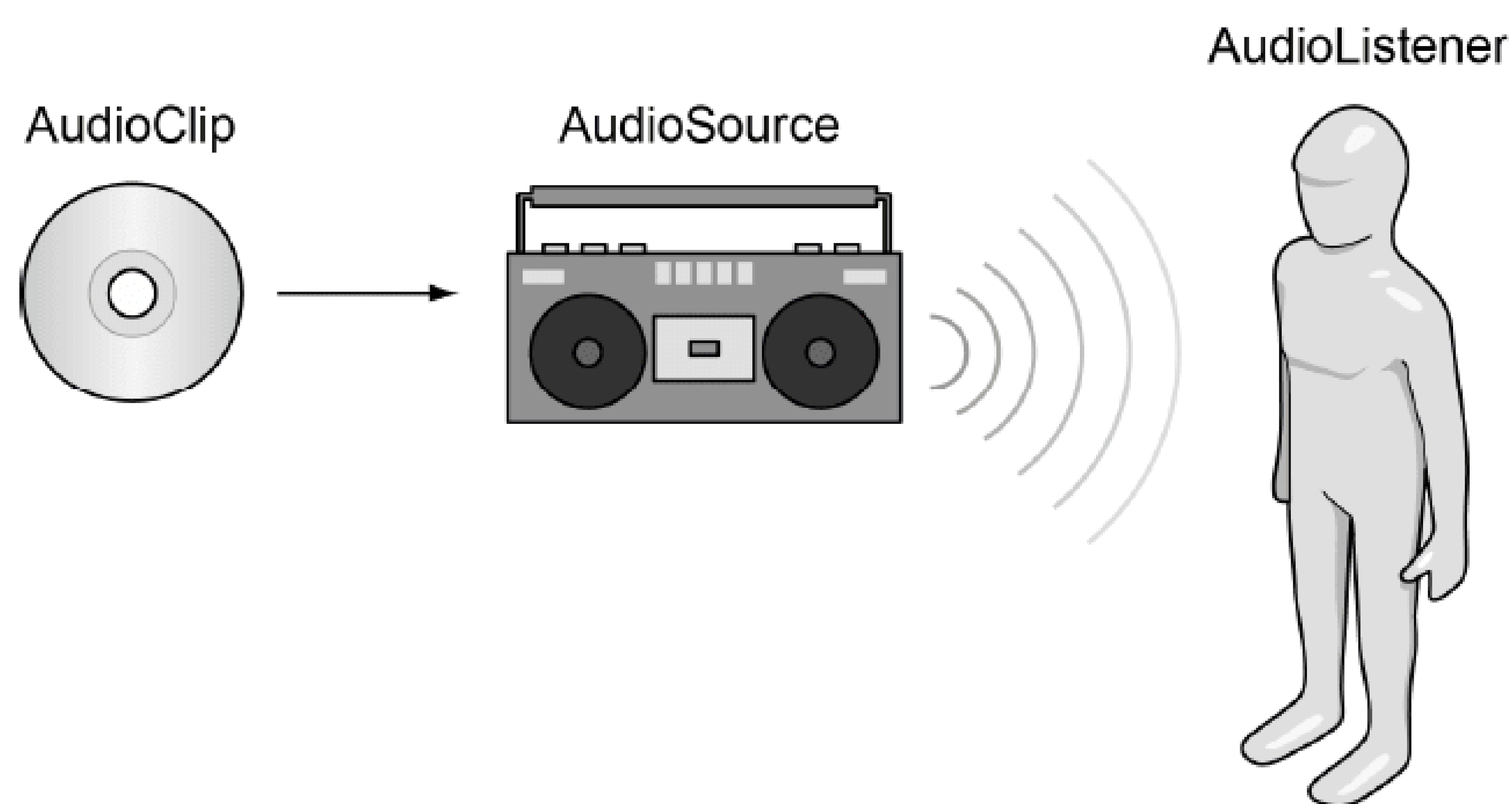


图 11-2　在 Unity 的音频系统中控制的三种对象

Unity 音频系统中包括的第三种对象是 AudioListener。顾名思义，这个对象听取音源投射的声音。这是关于声音系统作用的另一个抽象(显然，真正的听众是游戏的玩家！)，但——就像音源给出它投射的位置一样，音频侦听器指定它在哪个位置听取声音。

使用音频混合进行高级声音控制

音频混合是 Unity 5 中增加的一个新特性。音频混合不是直接播放音频剪辑，而是允许处理音频信号，并将不同的效果应用到剪辑中。要学习更多关于音频混合的知识，可以在 Unity 的文档中找到，例如，请观看如下这个辅导视频：http://mng.bz/Mlp3。

尽管必须设定音频剪辑和 AudioSource 组件，但创建新场景时，AudioListener 组件默认已存在于摄像机上。通常，需要通过 3D 音效响应场景中观察者的位置。

11.2.2　设定循环播放的声音

现在，在 Unity 中设置第一个声音！音频剪辑已经导入，而默认的摄像机有一个 AudioListener 组件，因此只需要设定一个 AudioSource 组件。接下来在 Enemy 预设(即四处走动的敌人角色)上放置噼里啪啦的开火声。

注意 当敌人在开火时会发出声音，可以给它指定一个粒子系统，以便它看起来像是着火了。把第 4 章的粒子系统变成一个预设，并从 Asset 菜单中选择 Export Package，就可以将该粒子对象复制到本示例项目中。也可以从头重做第 4 章所做的步骤(将空预设拖动到场景中并编辑它们，接着选择 GameObject | Apply Changes To Prefab)。

通常，为了编辑预设，需要将它拖到场景中。将组件添加到对象上时，也可以直接编辑预设资源。选择 Enemy 预设，使它的属性出现在 Inspector 中。现在添加一个新组件，选择 Audio | Audio Source。这样，AudioSource 组件便出现在 Inspector 中。

告诉音源要播放哪个声音剪辑。从 Project 视图将音频文件拖动到 Inspector 的 Audio Clip 槽上，本例使用“fireplace”音效(如图 11-3 所示)。

跳过一些设置，选择 Play On Awake 和 Looping(当然，要确保 Mute 没有被选中)。Play On Awake 告诉音源，在场景启动时开始播放声音(下一节将介绍如何在场景运行时手动触发声音)。Looping 告诉音源持续播放，当回放结束时重复播放声音剪辑。

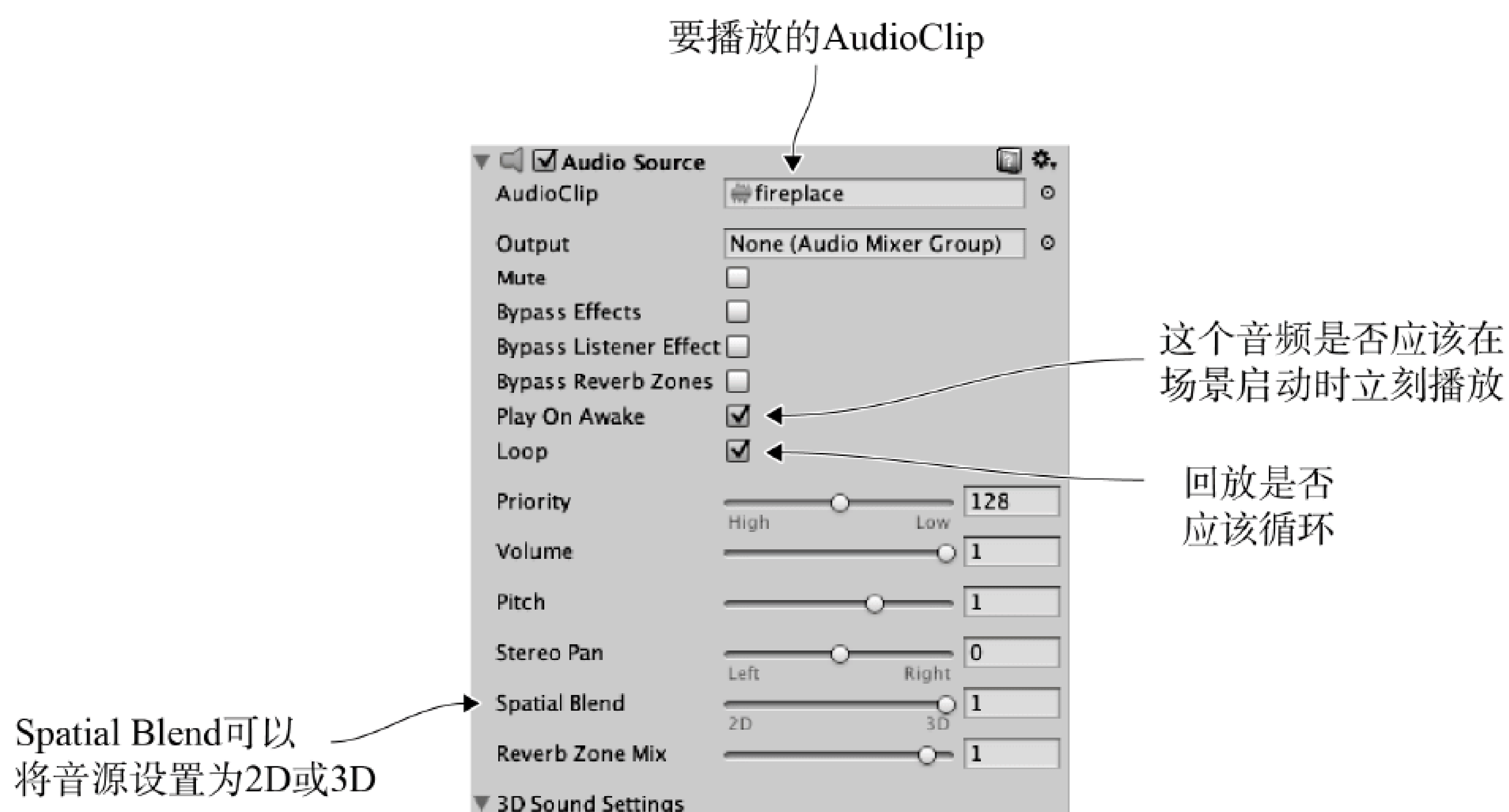


图 11-3　AudioSource 组件的设置

这个音源应播放 3D 声音。如前所述，3D 声音在场景中有具体的位置。音源使用 Spatial Blend 设置调整该位置。这个设置是 2D 和 3D 间的一个滑动条。将这个音源设置为 3D。

现在运行游戏并确保打开扬声器。可以听到来自敌人的噼里啪啦的开火声，而由于使用了 3D 音源，因此如果走开，则声音将变弱。

11.2.3　用代码触发音效

对于一些循环音效，设置 AudioSource 组件自动播放很便利，但大多数音效想要通过代码命令触发。这种方式依然需要 AudioSource 组件，但现在音源组件仅在由程序告知时播放声音，而不是一直自动播放。

将 AudioSource 组件添加到玩家对象上(不是摄像机对象)。不必将特定的音频剪辑链接到组件上，因为音频剪辑将在代码中定义。可以关闭 Play On Awake，因为这个音源的声音将通过代码触发。同时，将 Spatial Blend 调整为 3D 音效，因为该声音位于场景中。

现在，在处理射击的 RayShooter 脚本中添加代码清单 11.1 中的内容。

代码清单 11.1　RayShooter 脚本中添加的音效

```
...
[SerializeField] private AudioSource soundSource;
[SerializeField] private AudioClip hitWallSound;
[SerializeField] private AudioClip hitEnemySound;
...

if (target != null) {
    target.ReactToHit();
   soundSource.PlayOneShot(hitEnemySound);
} else {
    StartCoroutine(SphereIndicator(hit.point));
   soundSource.PlayOneShot(hitWallSound);
}
...
```

引用了要播放的两个声音文件

如果目标不为 null，玩家击中敌人，因此……

调用 PlayOneShot()播放 Hit An Enemy 声音，或者……

玩家未击中时，调用 PlayOneShot()播放 Hit A Wall 声音

新代码在脚本顶部包括了一些序列化变量。将玩家对象(带有 AudioSource 组件的对象)拖动到 Inspector 中的 soundSource 槽上。接着将要播放的音频剪辑拖动到 sound 槽上。“swish”是击中墙壁的声音，而“ding”是击中敌人的声音。

另外添加的两行是 PlayOneShot()方法。该方法使音源播放给定的音频剪辑。在 target 条件中添加的那两个方法是为了击中不同目标时播放不同的音效。

注意 可以在 AudioSource 中设置剪辑，调用 Play()来播放剪辑。多个声音必须彼此分开，因此，我们使用 PlayOneShot()。使用下面的代码替代 PlayOneShot()，并快速射击，看看有什么问题:

```
soundSource.clip = hitEnemySound; soundSource.Play();
```

运行游戏并四处射击。现在游戏中有了一些不同的音效。这些相同的步骤可以用于所有类型的音效。然而，游戏中健壮的声音系统不仅需要一系列分散的声音，至少，所有游戏都应该提供音量控制。接下来通过一个中心音频模块实现音量控制。

11.3　音频控制接口

继续前面章节建立的代码架构，接下来将创建一个 AudioManager。Managers 对象中有游戏使用的不同代码模块的主列表，例如用于玩家仓库的管理器。此时，创建音频管理器，并添加到那个列表中。这个中心音频模块允许调节游戏中的音频音量甚至关闭

它。最开始时只考虑音效，但在后续章节中，将扩展 AudioManager 以处理音乐。

11.3.1　建立中心 AudioManager

建立 AudioManager 的第一步是准备好 Managers 代码框架。复制第 10 章项目中的 IGameManager、ManagerStatus 和 NetworkService，这里不修改它们(记住，IGameManager 是所有管理器必须实现的接口，而 ManagerStatus 是由 IGameManager 使用的枚举。NetworkService 提供了对互联网的调用，而本章不会用到)。

注意 Unity 可能会提示一个警告，因为设定了 NetworkService 却没有使用它。可以忽略 Unity 的警告，我们希望代码框架可以访问互联网，尽管本章并不需要该功能。

另外，还要复制 Managers 文件，接下来为新的 AudioManager 调整它。现在先不处理它(或者如果轻微的编译错误让你抓狂，可以注释掉错误部分)。创建一个称为 AudioManager 的新脚本，该脚本可以由 Managers 代码引用(见代码清单 11.2)。

代码清单 11.2　AudioManager 的框架代码

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class AudioManager : MonoBehaviour, IGameManager {
   public ManagerStatus status {get; private set;}

   private NetworkService _network;

   // Add volume controls here (listing 11.4)

   public void Startup(NetworkService service) {
      Debug.Log("Audio manager starting...");

        _network = service;

      // Initialize music sources here (listing 11.11)

       status = ManagerStatus.Started;
   }
}
```

在此执行任何长时间运行的启动任务

如果有长时间运行的任务，将状态设置为 Initializing

这段初始代码像之前章节的管理器一样，它是实现 IGameManager 的类所需的最小代码量。现在可以使用新的管理器调整 Managers 脚本(见代码清单 11.3)。

代码清单 11.3　用 AudioManager 调整了的 Managers 脚本

```
using UnityEngine;
```

```
using System.Collections;
using System.Collections.Generic;

[RequireComponent(typeof(AudioManager))]

public class Managers : MonoBehaviour {
   public static AudioManager Audio {get; private set;}

   private List<IGameManager> _startSequence;

  void Awake() {
     Audio = GetComponent<AudioManager>();

       _startSequence = new List<IGameManager>();
       _startSequence.Add(Audio);

       StartCoroutine(StartupManagers());
   }

   private IEnumerator StartupManagers() {
       NetworkService network = new NetworkService();

       foreach (IGameManager manager in _startSequence) {
          manager.Startup(network);
       }

       yield return null;

       int numModules = _startSequence.Count;
       int numReady = 0;

       while (numReady < numModules) {
          int lastReady = numReady;
          numReady = 0;

          foreach (IGameManager manager in _startSequence) {
             if (manager.status == ManagerStatus.Started) {
                numReady++;
             }
          }

          if (numReady > lastReady)
             Debug.Log("Progress: " + numReady + "/" + numModules);

          yield return null;
       }

       Debug.Log("All managers started up");
   }
}
```

这个项目中只列出AudioManager，并没有列出PlayerManager等

如前面章节所述，在场景中创建 Game Managers 对象，并将 Managers 和 AudioManager 附加到这个空对象上。运行游戏，控制台中就显示了管理器的启动消息，但音频管理器还没有做任何事情。

11.3.2　音量控制 UI

AudioManager 的基础建立完毕，就该为它添加音量控制功能了。为了关闭音效或者调整音量，UI 会显示这些音量控制方法。

这里使用第 7 章介绍的新 UI 工具。具体而言，就是创建一个带按钮和滑动条的弹出窗口来控制音量设置(如图 11-4 所示)。下面列出大概步骤，没有探讨细节，如果需要复习，请参考第 7 章：

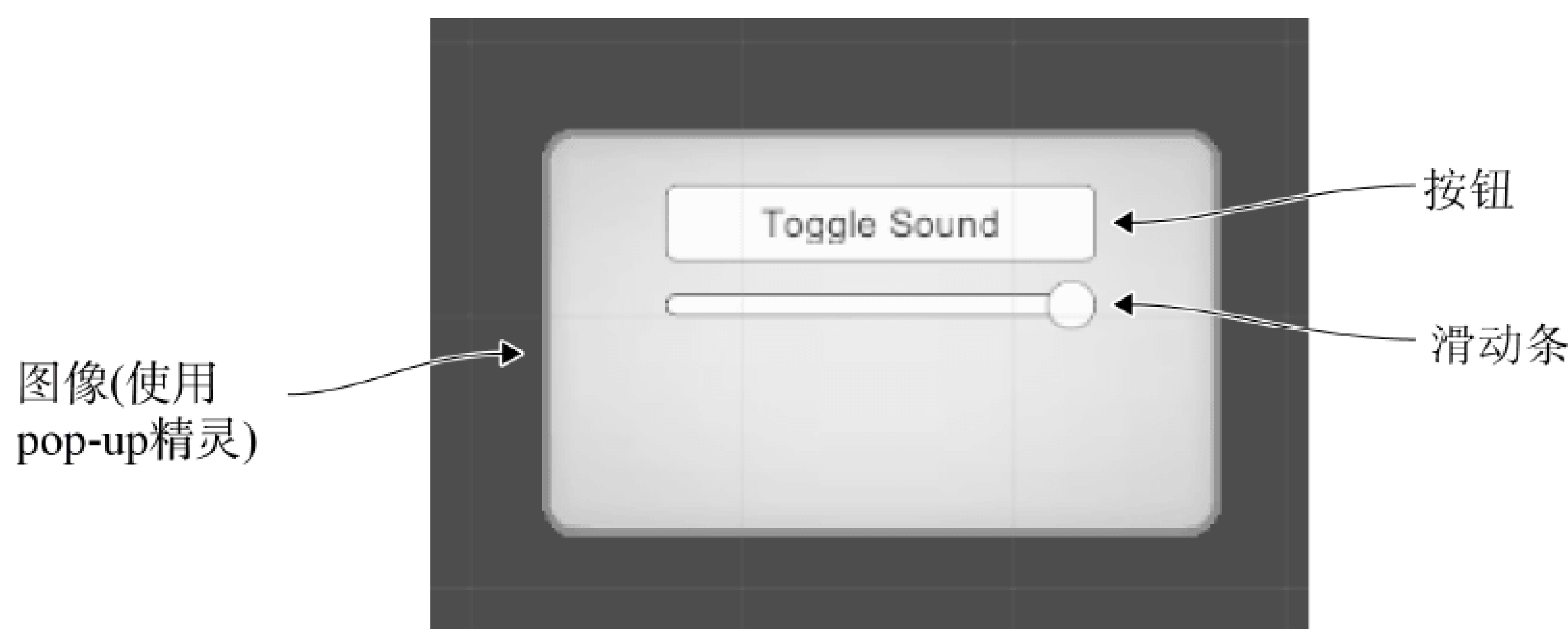


图 11-4　用于静音和音量控制的 UI

(1) 导入 popup.png 作为精灵(将 Texture Type 设置为 Sprite)。
(2) 在 Sprite Editor 中，将每条边设置为 12 像素(记住应用修改)。
(3) 在场景中创建画布(GameObject | UI | Canvas)。
(4) 为画布打开 Pixel Perfect 设置。
(5) (可选)将对象命名为 HUD Canvas，并切换为 2D 视图模式。
(6) 创建一个连接到画布的图像(GameObject | UI | Image)。
(7) 将新对象命名为 Settings Popup。
(8) 将 popup 精灵赋予到图像的 Source Image。
(9) 将 Image Type 设置为 Sliced 并打开 Fill Center。
(10) 将 pop-up 图像定位在(0, 0)，让它居中。
(11) 将 pop-up 缩放为 250 宽，150 高。
(12) 创建按钮(GameObject | UI | Button)。
(13) 使按钮的父节点为 pop-up(在 Hierarchy 中将按钮拖动到 pop-up 上)。
(14) 将按钮定位在(0, 40)。
(15) 展开按钮的层级，以便选择它的文本标签。
(16) 将文本修改为 Toggle Sound。
(17) 创建滑动条(GameObject | UI | Slider)
(18) 将滑动条的父结点设置为 pop-up 并定位在(0, 15)处。
(19) 将滑动条的 Value 设置为 1(在 Inspector 的底部)。

以上是创建设置弹出窗口的所有步骤！现在弹出窗口已经创建，接下来编写代码，

使它可以工作。这需要编写用于 pop-up 对象的脚本，以及 pop-up 脚本调用的音量控制功能。首先根据代码清单 11.4 调整 AudioManager 的代码。

代码清单 11.4　在 AudioManager 中添加音量控制

```
...
public float soundVolume {
    get {return AudioListener.volume;}
   set {AudioListener.volume = value;}
}

public bool soundMute {
   get {return AudioListener.pause;}
   set {AudioListener.pause = value;}
}

public void Startup(NetworkService service) {
   Debug.Log("Audio manager starting...");

   _network = service;

     soundVolume = 1f;

   status = ManagerStatus.Started;
}
...
```

带有 getter 和 setter 的音量属性

使用 AudioListener 实现 getter/setter

为静音添加一个类似的属性

斜体代码已经存在于脚本中，在此展示仅供参考

初始化值(0 到 1；1 是满音量)

把 soundVolume 和 soundMute 属性添加到 AudioManager 中。这两个属性的 get、set 函数使用 AudioListener 的全局值实现。AudioListener 类可以调整所有 AudioListener 实例收到的声音音量。设置 AudioManager 的 soundVolumn 属性与设置 AudioListener 的 volumn 有相同的效果。这里的优点在于封装：对音频的所有处理都通过一个管理器来处理，管理器外部的代码不需要了解所有的实现细节。

将这些方法添加到 AudioManager 后，就可以编写用于弹窗的脚本。创建一个称为 SettingsPopup 的脚本，并添加代码清单 11.5 中的内容。

代码清单 11.5　带有音量调整控件的 SettingsPopup 脚本

```
using UnityEngine;
using System.Collections;

public class SettingsPopup : MonoBehaviour {

   public void OnSoundToggle() {
      Managers.Audio.soundMute = !Managers.Audio.soundMute;
   }

     public void OnSoundValue(float volume) {
      Managers.Audio.soundVolume = volume;
    }
}
```

这个按钮将切换 AudioManager 中的静音属性

这个滑动条将调整 AudioManager 中的音量属性

该脚本中有两个方法影响 AudioManager 的属性：OnSoundToggle()设置 soundMute 属性，而 OnSoundValue()设置 soundVolume 属性。同往常一样，将 SettingsPopup 脚本拖动到 UI 中的 Settings Popup 对象上来链接 SettingsPopup 脚本。

接着，为了调用按钮和滑动条上的函数，把 pop-up 对象链接到这些控件的交互事件上。在按钮的 Inspector 中，找到标签为 OnClick 的面板。单击+按钮，为该事件添加一个新条目。将 Settings Popup 拖动到这个新条目的对象槽上，在菜单中找到 SettingsPopup，选择 OnSoundToggle()使按钮调用该函数。

这个方法也用于链接应用到滑动条的函数。首先查找滑动条设置面板的交互事件，本例中的面板称为 OnValueChanged。单击+按钮添加一个新条目，接着将 SettingsPopup 拖动到该对象槽中。在函数菜单中找到 SettingsPopup 脚本，在 Dynamic Float 下选择 OnSoundValue()。

警告 记住，在 Dynamic Float 下选择该函数而不是在 Static Parameter 下选择！尽管一个方法在列表中的两部分都会出现，但选择 Static Parameter 部分的方法只会收到一个提前输入的值(译者注：选择 Dynamic Float 部分的方法才会在每次收到最新的值)。

设置控件现在可以工作了，但还有另一个脚本需要处理：当前弹出窗口一直覆盖在屏幕上。一个简单的修复方法是让弹出窗口仅当按下 M 键时才打开。创建一个称为 UIController 的脚本，将该脚本链接到场景中的 Controller 对象上，并编写代码清单 11.6 中的代码。

代码清单 11.6　切换弹出设置的 UIController

```
using UnityEngine;
using System.Collections;

public class UIController : MonoBehaviour {
   [SerializeField] private SettingsPopup popup;    // 引用场景中的弹出窗口对象

   void Start() {
      popup.gameObject.SetActive(false);    // 初始化弹出窗口为隐藏
   }

   void Update() {
      if (Input.GetKeyDown(KeyCode.M)) {    // 使用 M 键切换弹出窗口
         bool isShowing = popup.gameObject.activeSelf;
         popup.gameObject.SetActive(!isShowing);

         if (isShowing) {
            Cursor.lockState = CursorLockMode.Locked;
            Cursor.visible = false;
         } else {
            Cursor.lockState = CursorLockMode.None;
            Cursor.visible = true;
         }    // 随着弹出窗口一起切换光标
```

```
            }
        }
    }
```

为了连接该对象引用，将弹出窗口拖动到脚本的对象槽上。运行并试着改变滑动条(记住，通过按下 M 键激活 UI)，同时四处开枪，听听声音效果，随着滑动条的移动，音效的音量也在改变。

11.3.3　播放 UI 声音

接下来为 AudioManager 添加另一个功能，允许单击按钮时播放 UI 声音。这个任务比它最初看起来更复杂，因为 Unity 需要 AudioSource。当场景中的对象发出声音时，在哪里需要添加音源是很明显的。但 UI 音效不是场景的一部分，因此要为 AudioManager 设立一个特殊的 AudioSource，在没有其他任何音源时使用。

创建一个空的 GameObject 并让它的父节点为 Game Managers 对象。这个新对象将拥有一个由 AudioManager 使用的 AudioSource，因此把新对象命名为 Audio。将 AudioSource 组件添加到这个对象上(这次让 Spatial Blend 保持为 2D，因为 UI 在场景中没有任何明确的位置)，接着在 AudioManager 中添加代码清单 11.7 中的代码，以使用这个音源。

代码清单 11.7　在 AudioManager 中播放音效

```
...
[SerializeField] private AudioSource soundSource;
...
public void PlaySound(AudioClip clip) {
   soundSource.PlayOneShot(clip);
}
...
```

Inspector 中的变量槽，用于引用新的音源

播放没有其他音源的声音

一个新的变量槽出现在 Inspector 中，将 Audio 对象拖动到这个槽上。现在将 UI 音效添加到弹出脚本上(见代码清单 11.8)。

代码清单 11.8　将音效添加到 SettingsPopup 中

```
...
[SerializeField] private AudioClip sound;
...
public void OnSoundToggle() {
   Managers.Audio.soundMute = !Managers.Audio.soundMute;
    Managers.Audio.PlaySound(sound);
}
...
```

Inspector 中引用声音剪辑的对象槽

当按下按钮时播放音效

将 UI 音效拖动到变量槽上，这里使用 2D 声音“thump”。当按下 UI 按钮，同时将播放音效(当然是在没有关闭声音的时候)。虽然 UI 本身没有任何音源，但 AudioManager 却有播放音效的音源。

建立了所有的音效后，现在开始关注音乐。

11.4　背景音乐

接下来将一些背景音乐添加到游戏中，为此要将音乐添加到 AudioManager 中。如本章引言所述，音乐剪辑和音效没有功能上的区别。数字音频通过波形的方式起作用的原理是相同的，而播放音频的命令大多也相同。主要的区别在于音频长度，但该区别导致了一些后果。

首先，音轨会消耗计算机的大量内存，而这种内存的消耗必须优化。必须小心内存方面的两个问题：在需要音乐前将其载入内存，载入时会消耗太多内存。

优化音乐的载入可以使用第 9 章介绍的 Resources.Load()命令。这个命令允许根据名称加载资源，虽然它确实是一个方便的特性，但它并非是从 Resources 目录加载资源的唯一原因。另一个关键的考虑是延迟加载。通常当场景载入时，Unity 会立刻加载场景中的所有资源，但 Resources 中的资源不会加载，除非手动获取它们。在本例中，音乐的音频剪辑采用懒惰加载的方式。另外，即使音乐未被使用，会浪费很多内存。

定义　懒惰加载是文件没有预先加载，而是直到需要时才加载。通常，如果在使用前加载数据，则数据响应会更快(例如，声音立刻播放)，但快速响应不太重要时，懒惰加载方式能节省很多内存。

第二个内存问题通过将音乐从磁盘中流式处理来解决。流式处理音频避免了计算机立即加载整个文件。这种加载方式在导入音频剪辑的 Inspector 中设置。

最后，还有一些步骤用于播放背景音乐，包括涵盖这些内存优化的步骤。

11.4.1　播放循环音乐

播放音乐的流程和播放 UI 音效系列步骤相同(背景音乐通常是场景中没有音源的 2D 声音)，因此接下来再次执行这些步骤：

(1) 导入音频剪辑。

(2) 建立供 AudioManager 使用的 AudioSource。

(3) 在 AudioManager 中编写代码，播放音频剪辑。

(4) 将音乐控制添加到 UI 上。

对每一步都做一些轻微的修改以处理音乐(而非音效)。下面介绍第一个步骤。

步骤(1)：导入音频剪辑

通过下载或录制音轨获取一些音乐。示例项目在 www.freesound.org/上下载了以下公共的循环音乐：

- “loop”由 Xythe/Ville Nousiainen 制作
- “Intro Synth”由 noirenex 制作

将这些文件拖到 Unity 中，导入它们，接着在 Inspector 中调整它们的导入设置。如前所述，音乐的音频剪辑通常和音效的音频剪辑有不同的设置。首先，音频格式应该设置为 Vorbis，即压缩音频。记住，压缩音频会显著减小文件的尺寸。压缩通常会轻微降低音频质量，但这种轻微的降低对于长音乐剪辑是可接受的，在出现的滑动条中设置 Quality 为 50%。

下一个调整的导入设置是 Load Type。同样，音乐应该从磁盘进行流处理而不是完全加载到内存。从 Load Type 菜单中选择 Streaming。类似的，打开 Load In Background，以便游戏在加载音乐时不会暂停或变慢。

实际上，调整完所有的导入设置之后，还必须将资源文件移动到正确的位置，以便正确加载。记住，Resources.Load()命令要求资源必须在 Resources 文件夹中。新建一个称为 Resources 的文件夹，在 Resources 文件夹中创建一个称为 Music 的文件夹，并将音频文件拖到 Music 文件夹中(如图 11-5 所示)。

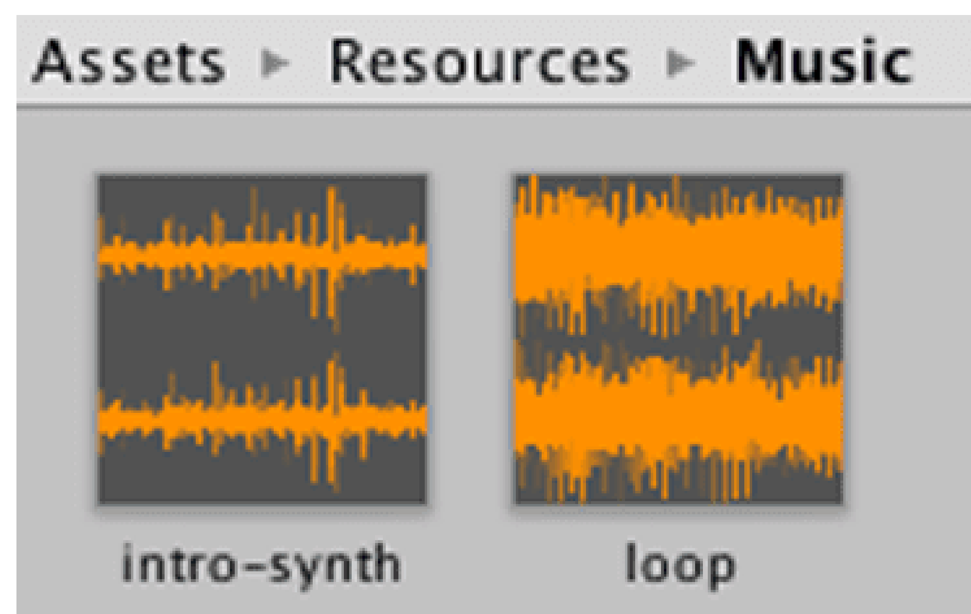


图 11-5　将音乐音频剪辑放在 Resources 文件夹中

以上是步骤(1)所完成的任务。

步骤(2)：建立一个用于 AudioManager 的 AudioSource

步骤(2)创建一个用于回放音乐的新 AudioSource。创建另一个空的 GameObject，命名为 Music 1(不是 Music，因为随后将添加 Music 2，并将其父节点调整为 Audio 对象。

将 AudioSource 组件添加到 Music 1 上，接着调整组件的设置。不要选择 Play On Awake，但这次需要打开 Loop 选项。音效通常播放一次，而音乐则是循环播放。让 Spatial Blend 设置保留为 2D，因为音乐在场景中没有特定的位置。

还要减小 Priority 值。对于音效，这个值无关紧要，因此使用其默认值 128。但对于音乐，就要减小这个值，因此设置音源为 60。这个值告诉 Unity，当分层多个声音时，哪个声音最重要。与直觉相反的是，越低的值有越高的优先级。当太多声音同时播放时，音频系统将丢弃一些声音，让音乐比音效具有更高的优先级。这样，同时触发太多音效时，可以确保一直在播放音乐。

步骤(3)：编写代码，在 AudioManager 中播放音频剪辑

现在，Music 音源已经建立完毕，将代码清单 11.9 中的内容添加到 AudioManager 中。

代码清单 11.9 在 AudioManager 中播放音乐

```
...
[SerializeField] private AudioSource music1Source;

[SerializeField] private string introBGMusic;        在这些字符串中
[SerializeField] private string levelBGMusic;        填写音乐名
...                                             从 Resources 加载 intro 音乐
public void PlayIntroMusic() {
   PlayMusic(Resources.Load("Music/"+introBGMusic) as AudioClip);
}                                                  从 Resources 加载主音乐
public void PlayLevelMusic() {
   PlayMusic(Resources.Load("Music/"+levelBGMusic) as AudioClip);
}

private void PlayMusic(AudioClip clip) {        通过设置 AudioSource.clip
   music1Source.clip = clip;                    属性播放音乐
   music1Source.Play();
}

public void StopMusic() {
   music1Source.Stop();
}
...
```

同往常一样，当选中 Game Managers 对象时，新的序列化变量将出现在 Inspector 中。将 Music 1 拖到音源槽中。接着在两个字符串变量中输入音乐文件的名称：intro-synth 和 loop。

剩余的新增代码调用命令来加载和播放音乐(在最后添加的方法中，是停止音乐的播放)。Resources.Load()命令从 Resources 文件夹中加载指定名称的资源(注意，音乐文件位于 Resources 中的 Music 子目录下)。Resources.Load()命令返回一个泛型对象，这个对象可以使用 as 关键字转换为更具体的类型(本例中，是 AudioClip)。

接着所加载的音频剪辑被传入 PlayMusic()方法中。这个方法设置了 AudioSource 的剪辑，接着调用 Play()。如前所述，音效最好使用 PlayOneShot()实现播放，但设置 AudioSource 的剪辑是使用音乐的一种更健壮的方式，允许停止或暂停正在播放的音乐。

步骤(4)：在 UI 上添加音乐控制

AudioManager 中新的音乐回放方法不会做任何事情，除非它们被其他代码调用。接下来将一些按钮添加到音频 UI，以便当按下按钮时能播放不同的音乐。下面再次列出了一些未详细解释的步骤(如果有需要，请参阅第 7 章)：

(1) 将弹出窗口的宽度修改为 350(以包含更多按钮)。

(2) 创建新 UI 按钮，并把它的父节点改为 pop-up。

(3) 将按钮的宽度设置为 100 并定位到(0, −20)。

(4) 展开按钮的层级并选择文本标签，将文本设置为 Level Music。

(5) 重复两次上述步骤，创建另外两个按钮。

(6) 将一个按钮定位在(-105, -20)，而另一个按钮定位在(105, -20)(因此它们会出现在两边)。

(7) 将第一个按钮的文本标签修改为 Intro Music，而另一个文本标签修改为 No Music。

现在 pop-up 有三个用于播放不同音乐的按钮。在 SettingsPopup 中编写一个方法(见代码清单 11.10)，它将链接到每个按钮。

代码清单 11.10　将音乐控制添加到 SettingsPopup 中

```
...
public void OnPlayMusic(int selector) {
     Managers.Audio.PlaySound(sound);

     switch (selector) {
     case 1:
       Managers.Audio.PlayIntroMusic();
       break;
   case 2:
       Managers.Audio.PlayLevelMusic();
       break;
   default:
       Managers.Audio.StopMusic();
       break;
   }
}
...
```

这个方法从按钮获取一个数字参数

对每个按钮调用 AudioManager 中不同的音乐方法

注意，这个方法这次带有一个 int 参数，通常按钮方法没有参数，它们只是由按钮触发。本例中，需要区分三个按钮，因此按钮发送一个不同的数字。

继续执行通常的步骤，将按钮连接到代码：在 Inspector 的 OnClick 面板上添加一个条目，将 pop-up 拖到对象槽中，并从菜单中选择相应的函数。这次，会出现一个文本框用于输入数字，因为 OnPlayMusic()带有一个数字参数。为 Intro Music 输入 1，为 Level Music 输入 2，而为 No Music 输入其他数字(作者输入 0)。OnMusic()中的 switch 语句根据该数字来播放 intro 音乐或 level 音乐，当数字不是 1 或 2 时停止播放。

在游戏运行时，按下音乐按钮，将听到音乐。代码从 Resources 目录中加载音频剪辑。音乐播放得很有效率，然而还有两个需要添加的优化：独立控制音乐的音量和当音乐切换时淡入淡出。

11.4.2　独立控制音乐的音量

游戏已经有音量控制，而当前也影响到音乐。大多数游戏对于音效和音乐的音量控制是分开的。因此接下来处理这个问题。

第一步是通知音乐的 AudioSource 忽略 AudioListener 的设置。需要全局 AudioListener 中的音量和静音继续影响所有的音效，但不想让这个音量作用到音乐上。代码清单 11.10

中包含的代码用于通知音源忽略 AudioListener 上的音量。代码清单 11.11 中的代码也添加了对音乐的音量控制和静音，因此将下面的代码添加到 AudioManager 中。

代码清单 11.11　在 AudioManager 中独立控制音乐的音量

```
...
private float _musicVolume;
public float musicVolume {
   get {
      return _musicVolume;
   }
   set {
      _musicVolume = value;

      if (music1Source != null) {
         music1Source.volume = _musicVolume;
      }
   }
}
...
public bool musicMute {
   get {
      if (music1Source != null) {
         return music1Source.mute;
      }
      return false;
   }
   set {
      if (music1Source != null) {
         music1Source.mute = value;
      }
   }
}

public void Startup(NetworkService service) {
   Debug.Log("Audio manager starting...");

   _network = service;

   music1Source.ignoreListenerVolume = true;
   music1Source.ignoreListenerPause = true;

   soundVolume = 1f;
   musicVolume = 1f;

   status = ManagerStatus.Started;
}
...
```

私有变量，不能直接访问，只通过属性的 getter 访问

直接调整 AudioSource 的音量

当 AudioSource 不存在时返回默认值

斜体代码已经存在于脚本中，在此展示仅供参考

这些属性通知 AudioSource 忽略 AudioListener 的音量

斜体代码已经存在于脚本中，在此展示仅供参考

这段代码的关键是可以直接调整 AudioSource 的音量，即使那个音源忽略定义在 AudioListener 中的全局音量。代码中有一些用于管理独立音源的音量和静音属性。

Startup()方法通过打开 ignoreListenerVolumn 和 ignoreListenerPause 初始化了音源。这些属性使音源忽略 AudioListener 中的全局音量设置。

现在可以运行游戏，验证音乐不再受已有音量控制的影响。接下来添加第二个UI，用于控制音乐音量。首先根据代码清单 11.12 调整 SettingsPopup。

代码清单 11.12 在 SettingsPopup 中控制音乐音量

```
...
public void OnMusicToggle() {
  Managers.Audio.musicMute = !Managers.Audio.musicMute;
Managers.Audio.PlaySound(sound);
}

public void OnMusicValue(float volume) {
  Managers.Audio.musicVolume = volume;
}
...
```

重复静音控制，仅使用 musicMute

重复音量控制，仅使用 musicVolume

这段代码没有太多解释——它主要重复音量控制。显然，AudioManager 使用的属性已经从 soundMute/soundVolume 变为 musicMute/musicVolumn。

在编辑器中，如之前所做创建一个按钮和滑动条。步骤如下：

(1) 将弹出窗口的高度改变为 225(以包含更多控件)。

(2) 创建 UI 按钮。

(3) 将按钮的父节点设置为 pop-up。

(4) 将按钮定位在(0, -60)。

(5) 展开按钮的层级，选择它的文本标签。

(6) 将文本改变为 Toggle Music。

(7) 创建一个滑动条(从相同的 UI 菜单中创建)。

(8) 将滑动条的父节点改为 pop-up，并定位在(0, -85)。

(9) 将滑动条的 Value(在 Inspector 底部)设置为 1。

将这些 UI 控件连接到 SettingsPopup 中的代码。在 UI 元素的设置中找到 OnClick/OnValueChanged 面板，单击+按钮添加一个条目，将 pop-up 对象拖到对象槽中，并从菜单中选择函数。需要选择的函数是菜单中 Dynamic Float 部分的 OnMusicToggle()和 OnMusicValue()。

现在运行代码，影响音效和音乐的控件分开了。这让音频系统变得相当精致，但还有一点需要优化：音轨间的淡入淡出。

11.4.3 歌曲间的淡入淡出

最后一个优化是，让 AudioManager 在不同背景的曲调间淡入淡出。当前在不同音轨间切换是很不协调的，声音突然终止，直接切换为新的音轨。使之前音轨的音量迅速降低到 0，新的音轨音量从 0 迅速增加，就可以实现平滑过渡。这是一段简单而巧妙的代码，它同时组合了前面的音量控制方法，通过协程随着时间逐步递增音量。

代码清单 11.13 将一些代码添加到 AudioMananger 中，但大多围绕一个简单的概念：现在有了两个独立的音源，在独立的音源中播放独立的音轨，在降低一个音源音量的同时递增另一个音源的音量(如往常一样，初始代码已经存在于脚本中，在此显示代码仅用于参考)。

代码清单 11.13　在 AudioManager 中对两个音乐进行淡入淡出处理

```
...
[SerializeField] private AudioSource music2Source;   ← 第二个 AudioSource (也保留第一个)

private AudioSource _activeMusic;     ← 记录哪个音源是激活的，哪个是非激活的
private AudioSource _inactiveMusic;

public float crossFadeRate = 1.5f;
private bool _crossFading;   ← 正在淡入淡出时用于避免 bug 的开关
...
public float musicVolume {
   ...
   set {
      _musicVolume = value;

      if (music1Source != null && !_crossFading) {
         music1Source.volume = _musicVolume;
         music2Source.volume = _musicVolume;   ← 调整两个音源的音量
         }
   }
}
...
public bool musicMute {
   ...
   set {
      if (music1Source != null) {
         music1Source.mute = value;
         music2Source.mute = value;
      }
   }
}

public void Startup(NetworkService service) {
   Debug.Log("Audio manager starting...");

   _network = service;

   music1Source.ignoreListenerVolume = true;
   music2Source.ignoreListenerVolume = true;
   music1Source.ignoreListenerPause = true;
   music2Source.ignoreListenerPause = true;
   soundVolume = 1f;
   musicVolume = 1f;

  _activeMusic = music1Source;         ← 初始化 1 为激活的 AudioSource
   _inactiveMusic = music2Source;

   status = ManagerStatus.Started;
```

```
}
...
private void PlayMusic(AudioClip clip) {
     if (_crossFading) {return;}
   StartCoroutine(CrossFadeMusic(clip));        当切换音乐时
}                                                调用协程
private IEnumerator CrossFadeMusic(AudioClip clip) {
   _crossFading = true;

   _inactiveMusic.clip = clip;
   _inactiveMusic.volume = 0;
   _inactiveMusic.Play();

   float scaledRate = crossFadeRate * _musicVolume;
   while (_activeMusic.volume > 0) {
      _activeMusic.volume -= scaledRate * Time.deltaTime;
      _inactiveMusic.volume += scaledRate * Time.deltaTime;

       yield return null;        该 yield 语句暂停一帧
   }

  AudioSource temp = _activeMusic;        用于交换_active 和
                                          _inactive 的临时变量
   _activeMusic = _inactiveMusic;
   _activeMusic.volume = _musicVolume;

   _inactiveMusic = temp;
   _inactiveMusic.Stop();

   _crossFading = false;
}

public void StopMusic() {
   _activeMusic.Stop();
   _inactiveMusic.Stop();
}
...
```

首先添加的是用于第二个音源的变量。保留第一个 AudioSource 对象，同时复制该对象(确保设置相同——选择 Loop)，并将新对象拖到这个 Inspector 槽中。这段代码也定义了 AudioSource 类型的变量 active 和 inactive，但这两个变量都是私有变量，未显示在 Inspector 中。具体而言，这些变量定义了哪个音源是“激活的”，哪个是“非激活的”。

现在当播放新音乐时，代码调用协程。这个协程设置新的音乐在一个 AudioSource 上播放，而旧音乐继续在旧音源上播放。接着协程逐步增加新音乐的音量，同时逐步降低旧音乐的音量。一旦淡入淡出完成(也就是音量完全交换过来)，这个函数就交换“激活的”和“非激活的”两个音源。

这就为游戏的音频系统实现了背景音乐。

FMOD：用于游戏音频的工具

Unity 中的音频系统由 FMOD 提供技术支持，这是一个有名的音频程序库。可以在 www.fmod.org 中找到这个库，现在它已经集成到 Unity 中。Unity 已经集成 FMOD 的很多特性，尽管如此，它还缺少这个库的最高级特性(可以浏览它们的网站，学习更多特性)。

这些高级音频特性通过 FMOD Studio 提供(一个添加了更多功能到 Unity 的插件)，但本章的示例仅使用 Unity 内置的功能。内置的核心功能包含游戏音频系统最重要的特性。大多数游戏开发者的音频需求都可以由这个插件的核心功能得以满足，但这个插件对于期望对游戏音频获得更复杂效果的开发者很有帮助。

11.5　小结

- 音效不该压缩而音乐应该压缩，但都应该使用 WAV 格式，因为 Unity 提供了对所导入音频的压缩。
- 音频剪辑可以是播放一致的 2D 声音，也可以是对侦听者位置做出响应的 3D 声音。
- 使用 Unity 的 AudioListener 可以很容易全局调节音效的音量。
- 可以设置播放音乐的独立音源的音量。
- 设置不同音源上的音量，可以让背景音乐淡入淡出。

第 12 章

将各部分整合为一个完整的游戏

本章涵盖：

- 从其他项目装配对象和代码
- 编程创建指向-单击的控件
- 将 UI 从旧系统升级为新系统
- 加载新关卡，响应目标
- 设置成功/失败条件
- 保存和加载玩家进度

本章的项目将把前面章节中的所有内容组合在一起。本书中的大部分章节都是相互独立的，没有用一个完整的游戏贯穿整本书。本章把之前分别介绍的内容组合起来，以学习如何通过所有这些分散部件来搭建一个完整的游戏。本章也会讨论游戏的外围结构，包括关卡的切换，游戏的结束(比如当游戏角色死亡时显示“Game Over”，或者到达出口时显示“Success”)。展示如何保存游戏的进度，因为随着游戏规模的扩大，保存玩家的游戏进度会变得越来越重要。

警告 本章会用到前面章节详细解释过的任务示例，所以只列出简略的步骤。如果你对某些步骤存有疑惑，可以参阅前面的相关章节来了解更多的细节，如第 7 章中有关 UI 的讲解。

本章的项目是一个动作 RPG 的演示游戏，在这类游戏中，摄像机放在较高的位置，向下俯视(如图 12-1 所示)，可以通过单击鼠标控制角色的移动，我们非常熟悉的 *Diablo* 就是一款动作 RPG 演示游戏。后面还会介绍另一种游戏类型，这样在本书结束时，你就对游戏种类有更多的了解。

本章的项目是本书中最大的游戏，它主要有以下功能：

- 可通过单击鼠标来控制角色移动的俯视图视角
- 可以通过单击来操作设备
- 可收集的散落物品
- 显示在 UI 窗口中的物件
- 当前关卡中四处游荡的敌人
- 可以保存游戏，恢复游戏的进度
- 必须按顺序完成的三个关卡

该项目的工作量很大，不过，这几乎是全书的最后一章了。

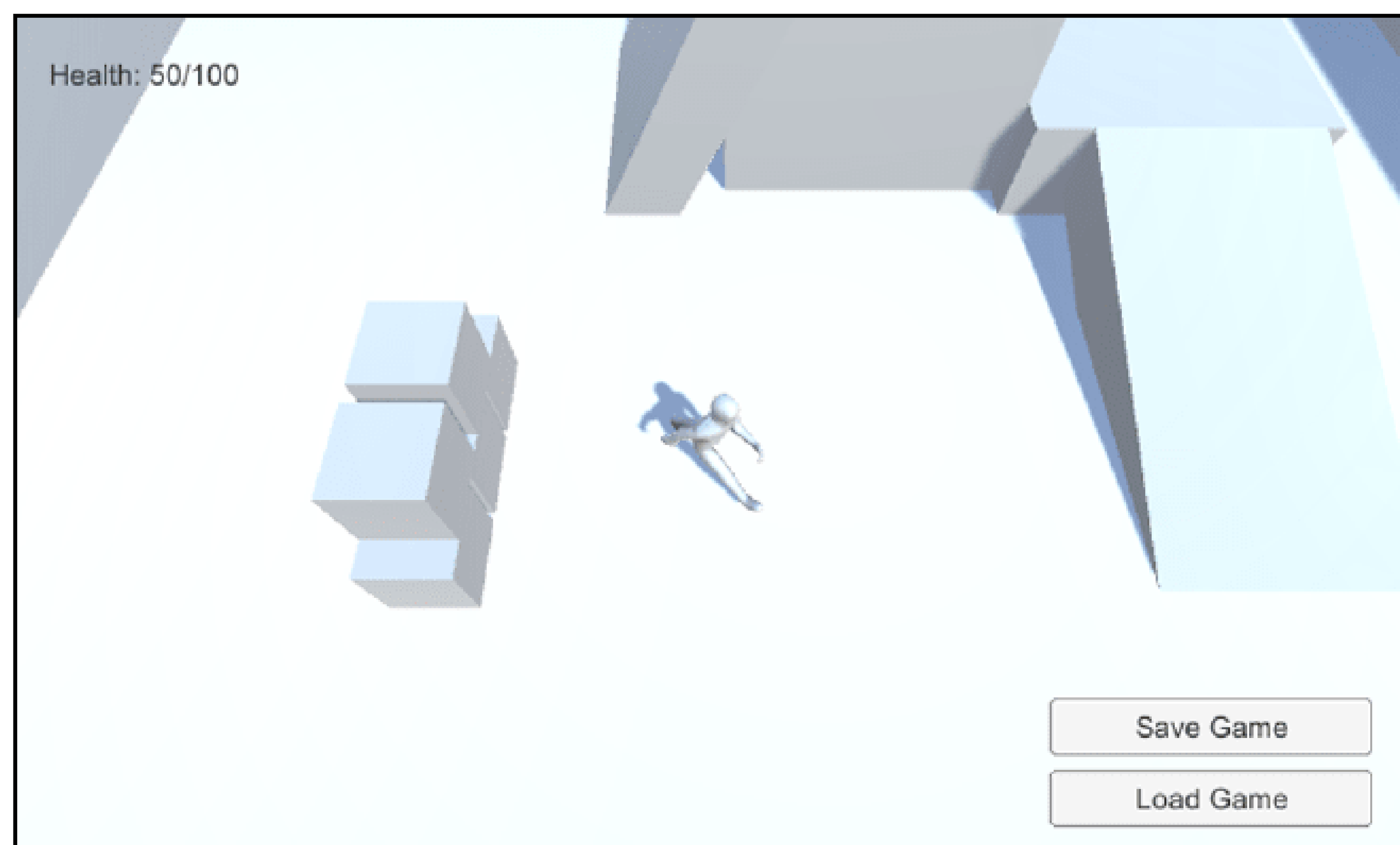


图 12-1　俯视视口的截图

12.1　再次利用项目构建动作 RPG 演示游戏

下面以第 9 章介绍的项目为基础来构建 RPG 演示游戏。复制该项目的文件夹，在 Unity 中打开副本，开始工作。如果直接跳到本章，则下载第 9 章的示例项目，然后在其基础上完成接下来的操作。

以第 9 章的项目为基础，是因为它和本章的目标最接近，需要的改动最少(与其他项目相比)。基本上，我们会将几章的资源组合在一起，所以从技术上来说，如果从其他章节开始，并入第 9 章的资源，则没有太大区别。

下面简述了第 9 章的项目：

- 一个已设定好动画控制器的角色
- 一个跟随该角色的第三人称视角摄像机
- 带有地面、墙壁和坡道的关卡
- 已设置好的灯光和阴影
- 可操作的设备，包括变色显示器
- 可收集的仓库物品
- 后端管理器代码框架

这个长功能列表包含了 PRG 演示游戏的许多功能，还有一些地方需要添加或修改。

12.1.1　将多个项目的资源和代码装配在一起

前两个修改是更新管理器框架并加入计算机控制的敌人。对于前一个任务，回想第 10 章对框架所做的更新，这意味着第 9 章的项目没有包含那些更新。对于后一个任务，回想第 3 章编写的敌人。

更新管理器框架

更新管理器是相当简单的，因此先完成这个任务。IGameManager 接口在第 10 章中修改了(见代码清单 12.1)。

代码清单 12.1　调整后的 IGameManager

```
public interface IGameManager {
   ManagerStatus status {get;}

   void Startup(NetworkService service);
}
```

代码清单 12.1 中的代码添加了对 NetworkService 的引用，因此也要确保复制那个额外的 NetworkService 脚本。将该文件从第 10 章的位置(记住，Unity 项目是磁盘上的一个文件夹，因此可以从文件夹中获得文件)拖放到新项目上。现在修改 Managers.cs，以使用修改过的接口(见代码清单 12.2)。

代码清单 12.2　稍微修改 Managers 脚本中的代码

```
...
private IEnumerator StartupManagers() {        ← 对该方法的开头
   NetworkService network = new NetworkService();   进行调整

   foreach (IGameManager manager in _startSequence) {
      manager.Startup(network);
}
...
```

最终，调整 InventoryManager 和 PlayerManager，以反映接口的变化。代码清单 12.3 展示了 InventoryManager 中修改的代码，对 PlayerManager 需要进行相同的代码修改，但使用不同的名称。

代码清单 12.3　调整 InventoryManager 以反映 IGameManager 的改变

```
...
private NetworkService _network;

public void Startup(NetworkService service) {
  Debug.Log("Inventory manager starting...");

  _network = service;

  _items = new Dictionary<string, int>();
  ...
```

对两个管理器进行同样的调整，但需要改变名称

一旦完成了所有次要的代码修改，所有对象就应和前面表现的一样。此处的更新应该看不出区别，游戏将和之前一样运行。这个调整很简单，但接下来的调整将比较复杂。

加入 AI 敌人

除了第 10 章调整的 NetworkServices 之外，还需要第 3 章的 AI 敌人。实现敌人角色需要加入一系列脚本和美术资源，因此需要导入所有这些资源。

首先复制这些脚本(记住，WanderingAI 和 ReactiveTarget 是用于 AI 敌人的行为脚本，Fireball 是发射的火焰，敌人攻击 PlayerCharacter 组件，SceneController 处理敌人的产生)：

- PlayerCharacter.cs
- SceneController.cs
- WanderingAI.cs
- ReactiveTarget.cs
- Fireball.cs

同样，拖入文件，以获得 Flame 材质、Fireball 预设和 Enemy 预设。如果从第 11 章而不是第 3 章获取敌人预设，就只需要再加入火焰粒子材质。

在复制所有需要的资源之后，资源间的连接可能会断开，因此为了使这些资源工作，需要重新连接这些资源。特别是要检查所有预设上的脚本，因为链接可能断开了。例如，Enemy 预设在 Inspector 中有两个丢失的脚本，因此单击圆圈按钮(如图 12-2 所示)并从脚本列表中选择 WanderingAI 和 ReactiveTarget。类似的，检查 Fireball 预设，重新连接需要的脚本。一旦处理好脚本，就检查到材质和贴图的连接。

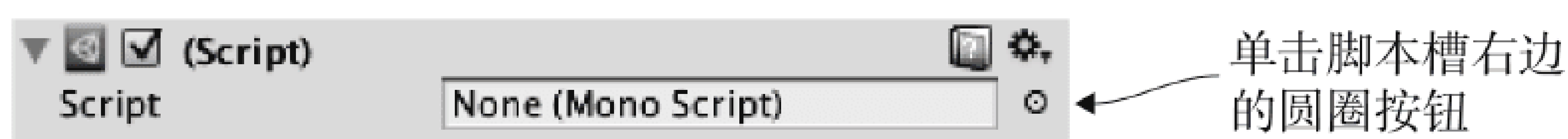


图 12-2　将脚本连接到组件

现在将 SceneController.cs 添加到控制器对象上，将 Enemy 预设拖到 Inspector 中的那个组件的 Enemy 槽上(选择 Enemy 预设，查看 Inspector 中的 WanderingAI)。另外，将 PlayerCharacter.cs 添加到玩家对象上，以便敌人攻击玩家。

运行游戏，敌人会四处移动。敌人将向玩家发射火球，但它不会造成大的伤害。选择 Fireball 预设，将 Damage 值设置为 10。

注意 当前，敌人还不是特别擅长追踪和击中玩家。本例首先给敌人指定更广的视野(使用第 9 章介绍的点积法)。我们要花费很多时间打磨游戏，包括迭代敌人的行为。打磨游戏可以使它更有趣，这虽然对于游戏的发布很重要，但不是本书要讨论的内容。

另一个问题是编写第 3 章的代码时，玩家的血量是一个用于测试的属性。现在游戏有了 PlayerManager，因此为了在该管理器中正常使用血量，根据代码清单 12.4 修改 PlayerCharacter。

代码清单 12.4　调整 PlayerCharacter，在 PlayerManager 中使用血量

```
using UnityEngine;
using System.Collections;

public class PlayerCharacter : MonoBehaviour {
  public void Hurt(int damage) {
    Managers.Player.ChangeHealth(-damage);   <-- 使用 PlayerManager 中的值而不是 PlayerCharacter 中的变量
  }
}
```

此时，游戏示例由多个之前的项目组合而成。敌人角色已添加到场景中，让游戏变得更加危险。但控件和视口依然来自第三人称移动游戏，因此接下来实现动作 RPG 的指向-单击控件。

12.1.2　编写指向-单击控件：移动和设备

这个示例需要俯视视角和玩家移动的鼠标控件(参见图 12-1)，而当前摄像机响应鼠标，玩家响应键盘(正同第 8 章所写的代码)，这和本章希望的效果相反。此外，还要修改变色显示器，以便通过单击来操作设备。在这两种情况下，现有的代码和希望的效果相差不是很远，接下来调整移动和设备脚本。

场景的俯视视角

首先，为了获得俯视视角，将摄像机的位置往上调到 Y 为 8。还要调整 OrbitCamera，以移除对摄像机的鼠标控制，仅使用方向键(见代码清单 12.5)。

代码清单 12.5　调整 OrbitCamera，移除鼠标控制

```
...
void LateUpdate() {
   _rotY -= Input.GetAxis("Horizontal") * rotSpeed;      ← 方向和之前相反
   Quaternion rotation = Quaternion.Euler(0, _rotY, 0);
   transform.position = target.position - (rotation * _offset);
   transform.LookAt(target);
}
...
```

摄像机的 Near/Far 裁剪平面

只要调整摄像机，就会提及 Near/Far 裁剪平面(clipping planes)。这些设置之前从未提及，因为默认设置工作得很好，但未来的项目可能需要调整这些设置。

选择场景中的摄像机，在 Inspector 中查看 Clipping Planes 部分，在其中可以输入 Near 和 Far 数字。这些值定义了渲染网格所在的近和远的边界：比 Near clipping plane 近的多边形或者比 Far clipping plane 远的多边形将不会绘制。

Near/Far 裁剪平面应相距足够远，以渲染场景中所有的物体，但它们应尽可能接近。当 Near/Far 裁剪平面相距太远时(Near 裁剪平面太近，而 Far 裁剪平面太远)，渲染算法就不再确定哪个多边形更近。这导致典型的渲染错误，称为 z-fighting(Z 轴用于表示深度)，两个多边形会彼此交叉。

随着摄像机升高，当运行游戏时，视角将向下。此时，移动控制依然使用键盘，因此接下来为指向-单击的移动编写脚本。

编写移动代码

这段代码(如图 12-3 所示)的基本理念是自动将玩家移动到目标位置。这个目标位置通过单击场景来设置。通过这种方式，移动玩家的代码不直接响应鼠标，而是通过单击来间接控制玩家的移动。

注意　这个移动算法也适用于 AI 角色。目标位置应该通过跟随角色的路径来设置，而不是通过鼠标单击来设置。



图 12-3　指向-单击控件的工作原理

为了实现这种控制，创建一个称为 PointClickMovement 的脚本，替换玩家上的 RelativeMovement 组件。开始编码 PointClickMovement 时，粘贴 RelativeMovement 的完整代码(因为依然需要处理下落和动画的大部分代码)。接着根据代码清单 12.6 来调整代码。

代码清单 12.6　PointClickMovement 脚本中的新移动代码

```
...
public class PointClickMovement : MonoBehaviour {
...
public float deceleration = 25.0f;
public float targetBuffer = 1.5f;
private float _curSpeed = 0f;
private Vector3 _targetPos = Vector3.one;

...
void Update() {
  Vector3 movement = Vector3.zero;

  if (Input.GetMouseButton(0)) {
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit mouseHit;
    if (Physics.Raycast(ray, out mouseHit)) {
      _targetPos = mouseHit.point;
      _curSpeed = moveSpeed;
    }
  }

  if (_targetPos != Vector3.one) {
    if (_curSpeed > moveSpeed * .5f) {
      Vector3 adjustedPos = new Vector3(_targetPos.x,
         transform.position.y, _targetPos.z);
      Quaternion targetRot = Quaternion.LookRotation(
         adjustedPos - transform.position);
      transform.rotation = Quaternion.Slerp(transform.rotation,
      targetRot, rotSpeed * Time.deltaTime);
}

  movement = _curSpeed * Vector3.forward;
  movement = transform.TransformDirection(movement);

  if (Vector3.Distance(_targetPos, transform.position) < targetBuffer) {
     _curSpeed -= deceleration * Time.deltaTime;
     if (_curSpeed <= 0) {
     _targetPos = Vector3.one;
    }
   }
}
_animator.SetFloat("Speed", movement.sqrMagnitude);
...
```

粘贴完代码后改正名称

当单击鼠标时设置目标位置

在鼠标位置发射射线

将目标位置设置为碰撞的位置

如果设置了目标位置，则移动

在快速移动时仅转向目标

当接近目标时减速到 0

此后所有情况保持一样的速度

Update()方法前面的大多数代码都比较容易理解，因为那些代码用于处理键盘控制的移动。注意，这些新代码有两个主要的 if 语句：一个在鼠标单击时运行，一个在

设置目标时运行。

当单击鼠标时，根据鼠标单击的位置来设置目标。这是射线发射的另一个用例：决定场景中的哪个点位于鼠标光标下。目标位置设置为鼠标击中的位置。

对于第二个条件语句，首先转向目标。Quaternion.Slerp()平滑旋转为面向目标，而不是突然面向目标。此时，减速(只以半速旋转)并锁定旋转(否则玩家在瞄准目标时旋转可能会很奇怪)。接着将向前移动的方向从玩家的当前坐标转换为全局坐标(以向前移动)。最后，检查玩家和目标之间的距离：如果玩家快到达目标，则降低移动速度，最终通过移除目标位置来结束移动。

练习：关闭跳跃控制

当前，这个脚本依然具备 RelativeMovement 中的跳跃控制。玩家依然可以在空格键按下时跳起，但指向-单击的移动中不应该有跳跃按钮。提示：调整'if (hitGround)'条件分支中的代码。

以上介绍了使用鼠标控件移动玩家。运行游戏，测试它。接下来介绍如何使用鼠标操作设备。

使用鼠标操作设备

在第 9 章中(直到在本章调整代码之前)，设备通过按键来操作。设备应该通过鼠标来操作。为此首先创建一个所有设备都可以继承的基础脚本，这个基础脚本将带有鼠标控制，而设备将继承它。创建一个称为 BaseDevice 的脚本，编写代码清单 12.7 中的代码。

代码清单 12.7　BaseDevice 脚本在鼠标单击时运行

```
using UnityEngine;
using System.Collections;

public class BaseDevice : MonoBehaviour {
  public float radius = 3.5f;

 void OnMouseDown() {
   Transform player = GameObject.FindWithTag("Player").transform;
   if (Vector3.Distance(player.position, transform.position) < radius) {
     Vector3 direction = transform.position - player.position;
     if (Vector3.Dot(player.forward, direction) > .5f) {
      Operate();
     }
   }
 }

 public virtual void Operate() {
   // behavior of the specific device
 }
}
```

单击时运行的函数

如果玩家在附近并面对它，则调用 Operate()

virtual 标记可以在继承时重写的方法

大多数代码都在 OnMouseDown()内，因为单击对象时，MonoBehaviour 会调用这个方法。首先，它检查玩家的距离，接着使用点积观察玩家是否面向设备。Operate()是一个空壳方法，这个方法的实现代码将由从 BaseDevice 继承的设备填充。

注意 这段代码在场景中查找标签(tag)为 Player 的对象，因此将 Player 标签赋予玩家对象。标签是 Inspector 顶部的下拉菜单，也可以自定义标签，但默认定义了一些标签，包括 Player 标签。选择玩家对象，编辑标签，接着选择 Player 标签。

现在 BaseDevice 已经编写好，可以修改 ColorChangeDevice 来继承 BaseDrvice 脚本。代码清单 12.8 展示了更新后的代码。

代码清单 12.8　调整 ColorChangeDevice，继承 BaseDrvice 脚本

```
using UnityEngine;
using System.Collections;

public class ColorChangeDevice : BaseDevice {
  public override void Operate() {
    Color random = new Color(Random.Range(0f,1f),
      Random.Range(0f,1f), Random.Range(0f,1f));
   GetComponent<Renderer>().material.color = random;
  }
}
```

继承 BaseDevice 而不是继承 MonoBehaviour

重写基类中的这个方法

由于这个脚本继承 BaseDevice 而不是继承 MonoBehaviour，因此它获得了鼠标控制的功能。接着重写空的 Operate()方法，以编写变色行为。

现在，当单击鼠标时，设备将被操作。同时移除玩家的 DeviceOperator 脚本组件，因为该脚本通过控制按键来操作设备。

这个新的设备输入和移动控件一起工作时存在一个问题：当前，移动目标随时根据鼠标单击而设置，但在单击设备时不希望设置移动目标。可以使用层(layer)来解决这个问题。类似在玩家上设置标签的方式，可以将对象设置为不同的层，代码可以检查那些层。调整 PointClickMovement，检查对象的层(见代码清单 12.9)。

代码清单 12.9　调整 PointClickMovement 中的鼠标单击代码

```
...
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
RaycastHit mouseHit;
if (Physics.Raycast(ray, out mouseHit)) {
  GameObject hitObject = mouseHit.transform.gameObject;
  if (hitObject.layer == LayerMask.NameToLayer("Ground")) {
    _targetPos = mouseHit.point;
    _curSpeed = moveSpeed;
  }
}
...
```

添加的代码，其他代码是为了便于参考

该代码清单在鼠标单击代码中添加了一个条件，以检查单击的对象是否在 Ground 层上。层(类似标签)是 Inspector 顶部的下拉菜单，单击它可以查看菜单选项。和标签一样，已经默认定义了一些层。若要创建新层，就在菜单中选择 Edit Layers。在空的层槽中输入 Ground(可能是 slot 8，代码中的 NameToLayer()将名称转换为层数，以便使用名称而不是数字)。

现在 Ground 层已添加到菜单中，设置地面对象为 Ground 层——这意味着建筑的地面、坡道和玩家可以站立的平台也设置为 Ground 层。选择这些对象，并在 Layers 菜单中选择 Ground。

运行游戏，当单击变色显示器时，玩家不会移动。很好，指向-单击的控件已经完成！从之前项目中引入本项目的另一个对象是 UI。

12.1.3　使用新界面替换旧 GUI

第 9 章使用的是 Unity 的旧立即模式 GUI，因为这种方式易于编码。但第 9 章的 UI 看起来不如第 7 章的漂亮，因此接下来引入该界面系统。新 UI 带来的视觉效果比旧 UI 更好。图 12-4 展示了要创建的界面。

首先，创建 UI 图形。一旦 UI 图像在场景中准备就绪，就可以将脚本附加到 UI 对象上。接下来列出的步骤均不涉及细节，如果需要回顾这些内容，请参阅第 7 章。

(1) 导入 popup.png 作为精灵(选择 Texture Type)。

(2) 在 Sprite Editor 中，所有边都设置 12 像素宽(记得应用修改)。

(3) 在场景中创建画布(GameObject | UI | Canvas)。

(4) 选择画布的 Pixel Perfect 设置。

(5) 可选：将对象命名为 HUD Canvas 并切换为 2D 视图模式。

(6) 创建一个连接到画布的 Text 对象(GameObject | UI | Text)。

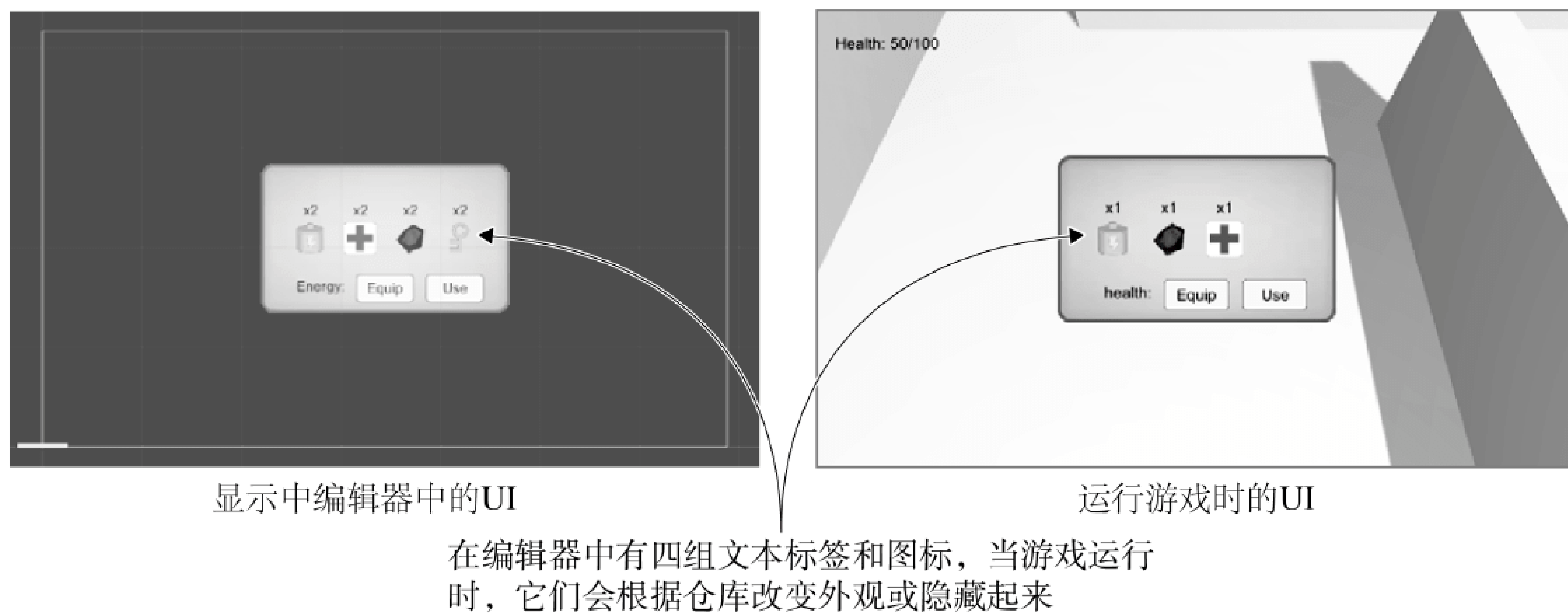


图 12-4　本章项目的 UI

(7) 将 Text 对象的锚点设置为左上角，定位为(100, -40)。
(8) 输入 Health:作为文本的标签。
(9) 创建连接到画布的图像(GameObject | UI | Image)。
(10) 将新对象命名为 Inventory Popup。
(11) 将弹出窗口精灵赋给图像的 Source Image。
(12) 将 Image Type 设置为 Sliced 并选择 Fill Center。
(13) 将弹出窗口图像定位在(0, 0)，将弹出窗口缩放为宽 250，高 150。

注意 回想如何在 3D 场景和 2D 界面间切换：切换到 2D 视图模式，双击 Canvas 或者 Building，放大该对象。

现在边角有了 Health 标签，中心有了很大的蓝色弹出窗口。接下来编写这些部分的程序。再深入 UI 功能。界面代码将使用和第 7 章一样的消息系统，因此复制 Messenger 脚本。之后，创建 GameEvent 脚本(见代码清单 12.10)。

代码清单 12.10　消息系统将使用的 GameEvent 脚本

```
public static class GameEvent {
  public const string HEALTH_UPDATED = "HEALTH_UPDATED";
}
```

现在只定义了一个事件，但本章将添加更多的事件。从 PlayerManager.cs 中(见代码清单 12.11)广播这个事件。

代码清单 12.11　从 PlayerManager.cs 中广播血量事件

```
...
public void ChangeHealth(int value) {
health += value;
   if (health > maxHealth) {
     health = maxHealth;
   } else if (health < 0) {
     health = 0;
    }

   Messenger.Broadcast(GameEvent.HEALTH_UPDATED);   ← 在函数末尾添加这一行
}
...
```

每次在 ChangeHealth()结束时都会广播这个事件，用于告知其他程序，血量已经发生了变化。下面调整 Health 标签，以响应这个事件，因此创建一个 UIController 脚本(见代码清单 12.12)。

代码清单 12.12　UIController 脚本，用于处理界面

```
using UnityEngine;
```

```
using UnityEngine.UI;
using System.Collections;

public class UIController : MonoBehaviour {
   [SerializeField] private Text healthLabel;          ← 引用场景中的 UI 对象
   [SerializeField] private InventoryPopup popup;

   void Awake() {          ← 设置血量更新事件的侦听器
      Messenger.AddListener(GameEvent.HEALTH_UPDATED, OnHealthUpdated);
   }
   void OnDestroy() {
      Messenger.RemoveListener(GameEvent.HEALTH_UPDATED,OnHealthUpdated);
   }

   void Start() {
      OnHealthUpdated();          ← 启动时手动调用函数

      popup.gameObject.SetActive(false);          ← 将弹出窗口初始化为隐藏
   }

   void Update() {
      if (Input.GetKeyDown(KeyCode.M)) {          ← 使用 M 键开关弹出窗口
         bool isShowing = popup.gameObject.activeSelf;
         popup.gameObject.SetActive(!isShowing);
         popup.Refresh();
      }
   }
                                          ← 事件侦听器调用函数，更新 Health 标签
   private void OnHealthUpdated() {
      string message = "Health: " + Managers.Player.health + "/" +Managers.
     Player.maxHealth;
      healthLabel.text = message;
   }
}
```

将这个新脚本附加到 Controller 对象并移除 BasicUI。另外，创建 InventoryPopup 脚本(现在添加空的公有方法 Refresh()，剩余代码在之后填充)，附加到弹出窗口上(Image 对象)。现在可以将弹出窗口拖到 Controller 组件的引用槽上，也将 Health 标签连接到 Controller 上。

玩家受到伤害或使用血量包时，Health 标签会变化，可以按下 M 键开关弹出窗口。最后一个要调整的细节是当前单击弹出窗口将导致玩家移动，与对设备的处理一样，不希望在单击 UI 时设置目标位置。代码清单 12.13 对 PointClickMovement 进行了调整。

代码清单 12.13　在 PointClickMovement 中检查 UI

```
using UnityEngine.EventSystems;
...
void Update() {
   Vector3 movement = Vector3.zero;
   if (Input.GetMouseButton(0) && !EventSystem.current.
     IsPointerOverGameObject()) {
   ...
```

注意，上面的条件检查鼠标是否在 UI 上。以上完成了界面的整体结构，因此接下来实现仓库弹出窗口。

实现仓库弹出窗口

弹出窗口当前是空白的，但它应该显示玩家的仓库(如图 12-5 所示)。以下步骤将创建 UI 对象：

(1) 创建四个图像，并使弹出窗口成为它们的父节点(即在 Hierarchy 中拖动对象)。

(2) 创建四个文本标签，使弹出窗口成为它们的父节点。

(3) 将所有图像定位在 Y 为 0，X 分别为−75, −25, 25, 75。

(4) 将文本标签定位在 Y 为 50，X 分别为−75, −25, 25, 75。

(5) 将文本(不是锚点)设置为 Center 对齐，Bottom 垂直对齐，Height 为 60。

(6) 在 Resources 目录中，将所有的仓库物品图标设置为 Sprite(而不是 Textures)。

(7) 将这些精灵拖到 Image 对象的 Source Image 槽(同时设置 Native Size)。

(8) 为所有文本标签输入 x2。

(9) 添加另一个文本标签和两个按钮，都设置其父节点为弹出窗口。

(10) 将这个文本标签定位在(−120, −55)并设置为 Right 对齐。

(11) 为这个文本的标签输入 Energy:。

(12) 将所有的按钮设置为 Width 60，接着定位在 Y 为−50, X 为 0 或 70。

(13) 在一个按钮上输入 Equip，在另一个按钮上输入 Use。

这些是仓库弹出窗口的可视化元素。将代码清单 12.14 中的内容写入 InventoryPopup 脚本中。

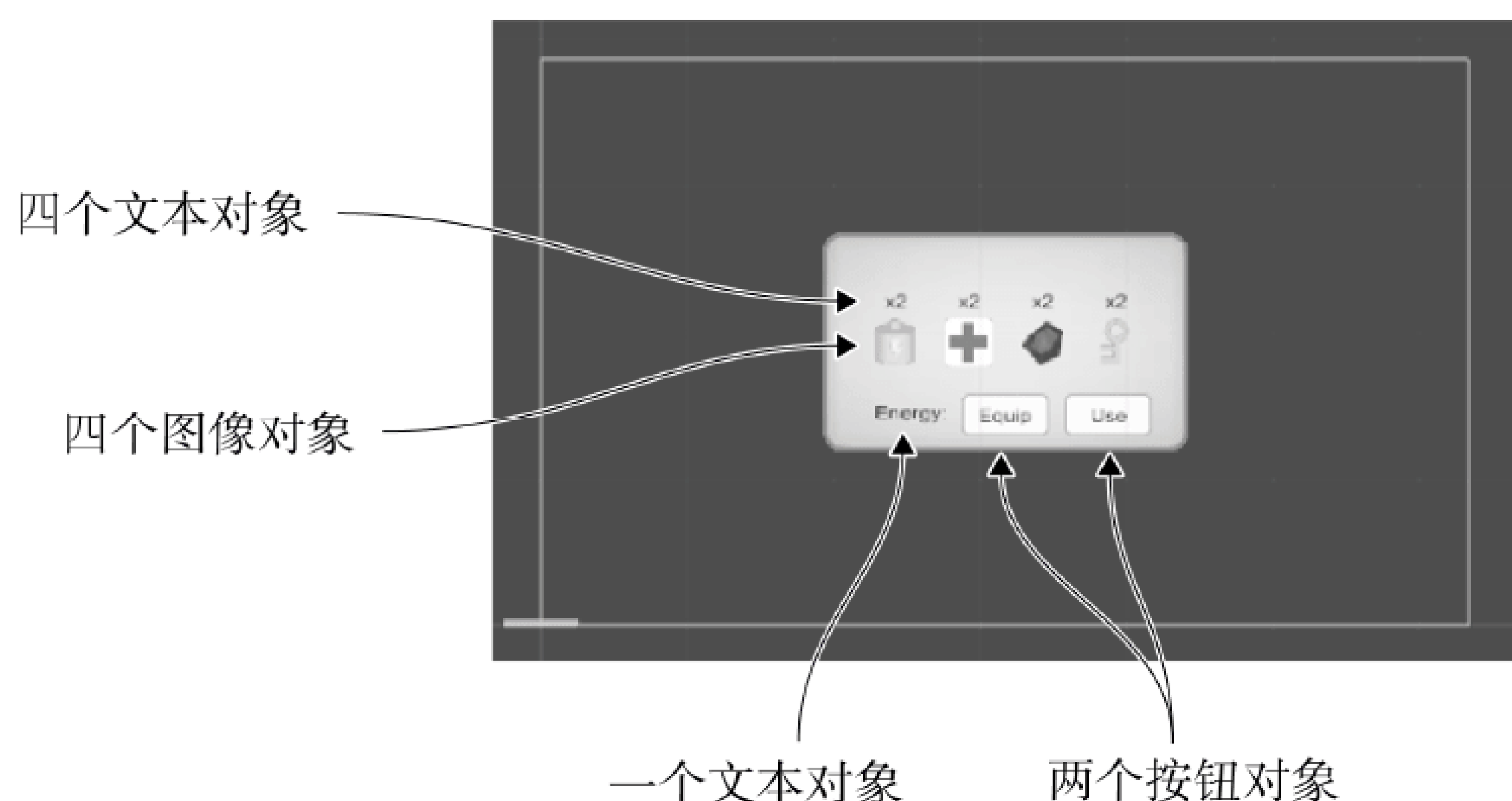


图 12-5　仓库 UI 图

代码清单 12.14　InventoryPopup 的完整脚本

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;
using System.Collections;
using System.Collections.Generic;
```

```
public class InventoryPopup : MonoBehaviour {
   [SerializeField] private Image[] itemIcons;          引用四个图像和
   [SerializeField] private Text[] itemLabels;          文本标签的数组

   [SerializeField] private Text curItemLabel;
   [SerializeField] private Button equipButton;
   [SerializeField] private Button useButton;

   private string _curItem;

   public void Refresh() {
      List<string> itemList = Managers.Inventory.GetItemList();

      int len = itemIcons.Length;
      for (int i = 0; i < len; i++) {     当循环所有 UI 图像时检查仓库列表
         if (i < itemList.Count) {
            itemIcons[i].gameObject.SetActive(true);
            itemLabels[i].gameObject.SetActive(true);

            string item = itemList[i];                    从 Resources 中加载精灵

            Sprite sprite = Resources.Load<Sprite>("Icons/"+item);

            itemIcons[i].sprite = sprite;
            itemIcons[i].SetNativeSize();     将图像的大小重新设置
                                              为精灵的原始大小
            int count = Managers.Inventory.GetItemCount(item);
            string message = "x" + count;
            if (item == Managers.Inventory.equippedItem) {
               message = "Equipped\n" + message;       标签除了显示物品数量外，
            }                                          还可能显示“Equipped”
            itemLabels[i].text = message;

            EventTrigger.Entry entry = new EventTrigger.Entry();      允许单
            entry.eventID = EventTriggerType.PointerClick;            击图标
            entry.callback.AddListener((BaseEventData data) => {
               OnItem(item);        为每个物品触发
            });                     不同的 lambda 函数

            EventTrigger trigger =
               itemIcons[i].GetComponent<EventTrigger>();
            trigger.triggers.Clear();                        清除侦听器，以便
            trigger.triggers.Add(entry);                     从空白的状态刷新
         }                                  将侦听器函数添
         else {                             加到 EventTrigger
            itemIcons[i].gameObject.SetActive(false);     如果没有物品需要显示，
            itemLabels[i].gameObject.SetActive(false);    则隐藏这个图像/文本
         }
      }

      if (!itemList.Contains(_curItem)) {
         _curItem = null;
      }                                    如果没有选择物品，则隐
      if (_curItem == null) {              藏按钮
         curItemLabel.gameObject.SetActive(false);
```

```
            equipButton.gameObject.SetActive(false);
            useButton.gameObject.SetActive(false);
        }
        else {
            curItemLabel.gameObject.SetActive(true);
            equipButton.gameObject.SetActive(true);
            if (_curItem == "health") {
                useButton.gameObject.SetActive(true);
            } else {
                useButton.gameObject.SetActive(false);
            }

            curItemLabel.text = _curItem+":";
        }
    }

    public void OnItem(string item) {
        _curItem = item;
        Refresh();
    }

    public void OnEquip() {
        Managers.Inventory.EquipItem(_curItem);
        Refresh();
    }

    public void OnUse() {
        Managers.Inventory.ConsumeItem(_curItem);
        if (_curItem == "health") {
            Managers.Player.ChangeHealth(25);
        }
    Refresh();
    }
}
```

显示当前选择的物品

仅使用按钮显示血量

由鼠标单击侦听器调用的函数

在改变物品后刷新仓库显示

好长的脚本！代码编写完毕后，在界面中将其他对象连接在一起。这个脚本组件现在包含各种对象引用，包括两个数组，展开这两个数组，并设置长度为 4(如图 12-6 所示)。将四个图像拖到 icons 数组上，将四个文本标签拖到 labels 数组上。

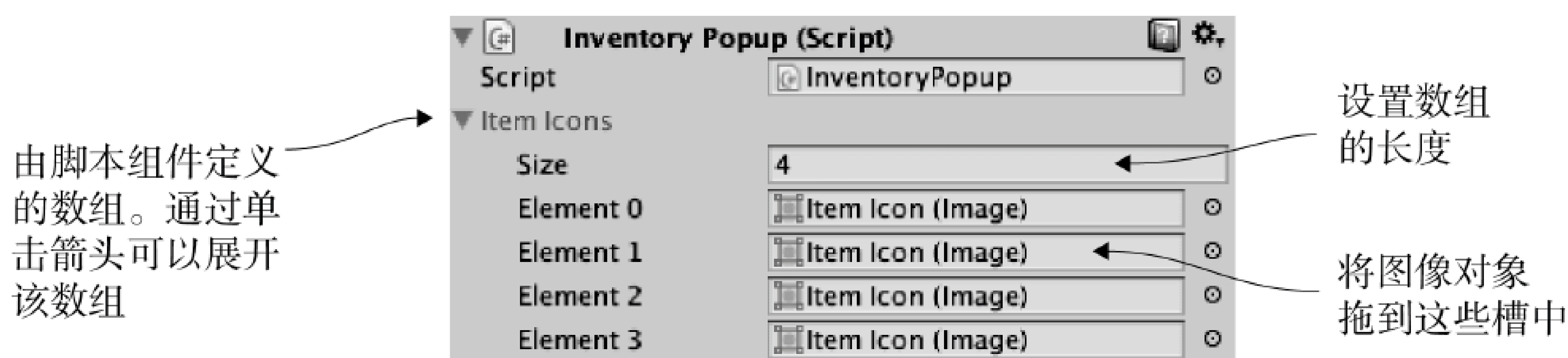


图 12-6　显示在 Inspector 中的数组

注意　如果不确定哪个对象连接到哪里(它们看起来都一样)，可以单击 Inspector 中的槽，查看 Hierarchy 视图中高亮显示的对象。

类似地，组件中的槽引用了弹出窗口下方的文本标签和按钮。在连接这些对象之

后，对这两个按钮添加 OnClick 侦听器。将这些事件连接到弹窗对象上，并相应地选择 OnEquip()或 OnUse()。

最终，将 EventTrigger 组件添加到所有四个物品图像上。InventoryPopup 脚本修改每个图标上的 EventTrigger 组件，因此它们最好有这个组件！在 Add Component | Event 下找到 EventTrigger(通过单击组件顶角的小齿轮按钮来复制/粘贴组件可能会更方便：从一个对象选择 Copy Component，在另一个对象上选择 Paste As New)。添加这个组件但不赋予事件侦听器，因为该操作已在 InventoryPopup 代码中完成了。

这就完成了仓库 UI 的处理！运行游戏，当收集物品并单击按钮时，观察弹出的仓库窗口。现在我们已完成了从前面项目中装配各部分内容，接下来将从这个项目开始讲解如何构建更复杂的游戏。

12.2　开发总体的游戏结构

现在有了一个有效的 RPG 游戏示例，下面将构建这个游戏的总体结构。这意味着构建游戏的整体流程，包括多个关卡和打通关卡的进度。第 9 章的项目只制作了一个关卡，但本章的路线图包括三个关卡。

此处理包括从 Managers 后台中进一步解耦场景，因此要广播与管理器相关的消息(与 PlayerManager 广播血量更新一样)。创建称为 StartupEvent(见代码清单 12.15)的脚本。在独立的脚本中定义这些事件，因为这些事件和可重用的 Managers 系统一起工作，而 GameEvent 仅针对这个游戏。

代码清单 12.15　StartupEvent 脚本

```
public static class StartupEvent {
   public const string MANAGERS_STARTED = "MANAGERS_STARTED";
   public const string MANAGERS_PROGRESS = "MANAGERS_PROGRESS";
}
```

现在开始调整 Managers，包括广播这些新事件！

12.2.1　控制任务流和多个关卡

当前，项目只有一个场景，Game Managers 对象就位于该场景中。问题是：每个场景都有自己的游戏管理器集合，然而所有场景都应共享一个游戏管理器。为此，创建独立的 Startup 场景，它将初始化管理器，并和游戏的其他场景共享该对象。

还需要一个处理游戏进程的新管理器。创建一个称为 MissionManager 的新脚本(见代码清单 12.16)。

代码清单 12.16　创建 MissionManager

```
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections;
using System.Collections.Generic;

public class MissionManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  public int curLevel {get; private set;}
  public int maxLevel {get; private set;}

  private NetworkService _network;

  public void Startup(NetworkService service) {
    Debug.Log("Mission manager starting...");

    _network = service;
    curLevel = 0;
    maxLevel = 1;

    status = ManagerStatus.Started;
  }

  public void GoToNext() {
     if (curLevel < maxLevel) {    ← 检查是否到达最后一个关卡
       curLevel++;
       string name = "Level" + curLevel;
       Debug.Log("Loading " + name);
       SceneManager.LoadScene(name);    ← 加载场景的 Unity 命令
     } else {
      Debug.Log("Last level");
     }
  }
```

代码清单 12.16 中的大部分代码并没有什么特殊之处，但要注意接近末尾的 LoadScene()方法。尽管在之前(第 5 章)提起过这个方法，但现在它很重要。这个 Unity 方法用于加载场景文件，第 5 章使用它在游戏中重新加载场景，还可以通过传入场景文件的名称来加载任何场景。

将这个脚本附加到场景中的 Game Managers 对象上。同时给 Managers 脚本添加一个新组件(见代码清单 12.17)。

代码清单 12.17　给 Managers 脚本添加一个新组件

```
...
[RequireComponent(typeof(MissionManager))]

public class Managers : MonoBehaviour {
  public static PlayerManager Player {get; private set;}
```

```
    public static InventoryManager Inventory {get; private set;}
    public static MissionManager Mission {get; private set;}
    ...
    void Awake() {
      DontDestroyOnLoad(gameObject);         // Unity 的命令，用于让对象在场景之间持久化

      Player = GetComponent<PlayerManager>();
      Inventory = GetComponent<InventoryManager>();
      Mission = GetComponent<MissionManager>();

      _startSequence = new List<IGameManager>();
      _startSequence.Add(Player);
      _startSequence.Add(Inventory);
      _startSequence.Add(Mission);

      StartCoroutine(StartupManagers());
  }

  private IEnumerator StartupManagers() {
    ...
      if (numReady > lastReady) {
        Debug.Log("Progress: " + numReady + "/" + numModules);
        Messenger<int, int>.Broadcast(
          StartupEvent.MANAGERS_PROGRESS,
           numReady, numModules);         // Startup 事件广播与事件相关的数据
      }

      yield return null;
    }

    Debug.Log("All managers started up");
    Messenger.Broadcast(StartupEvent.MANAGERS_STARTED);   // Startup 事件广播时不使用参数
  }
  ...
```

大多数代码都应该很熟悉了(添加 MissionManager 和添加其他管理器一样)，但还有两部分新代码。一部分是发送两个整数值的事件，之前介绍了两个泛型事件和带单一数字的消息，也可以使用相同的语法来发送任意数量的值。

另一部分新代码是 DontDestroyOnLoad()方法。它是由 Unity 提供的方法，用于在场景间持久化对象。通常场景中的所有对象会随着新场景的加载而被清除，但对对象使用 DontDestroyOnLoad()，可以确保该对象依然保留在新场景中。

用于启动和关卡的不同场景

由于 Game Managers 对象将在所有场景中持久化，因此必须将管理器从游戏关卡中独立出来。在 Project 视图中，复制场景文件(Edit | Duplicate)，并相应重命名两个文

件：一个为 Startup，另一个为 Level1。打开 Level1，删除 Game Managers 对象(它将由 Startup 提供)。打开 Startup，删除除 Game Managers、Controller、HUD Canvas 和 EventSystem 外的其他对象。为了调整摄像机，删除 OrbitCamera 组件，把 Clear Flags 菜单从 Skybox 改为 Solid Color。移除 Controller 上的脚本组件，并删除父节点为 Canvas 的 UI 对象(Health 标签和 InventoryPopup)。

这个 UI 当前是空的，因此创建一个新的滑动条(如图 12-7 所示)，关闭它的 Interactable 设置。Controller 对象不再有脚本组件，因此创建新的 StartupController 脚本，附加到 Controller 对象上(见代码清单 12.18)。

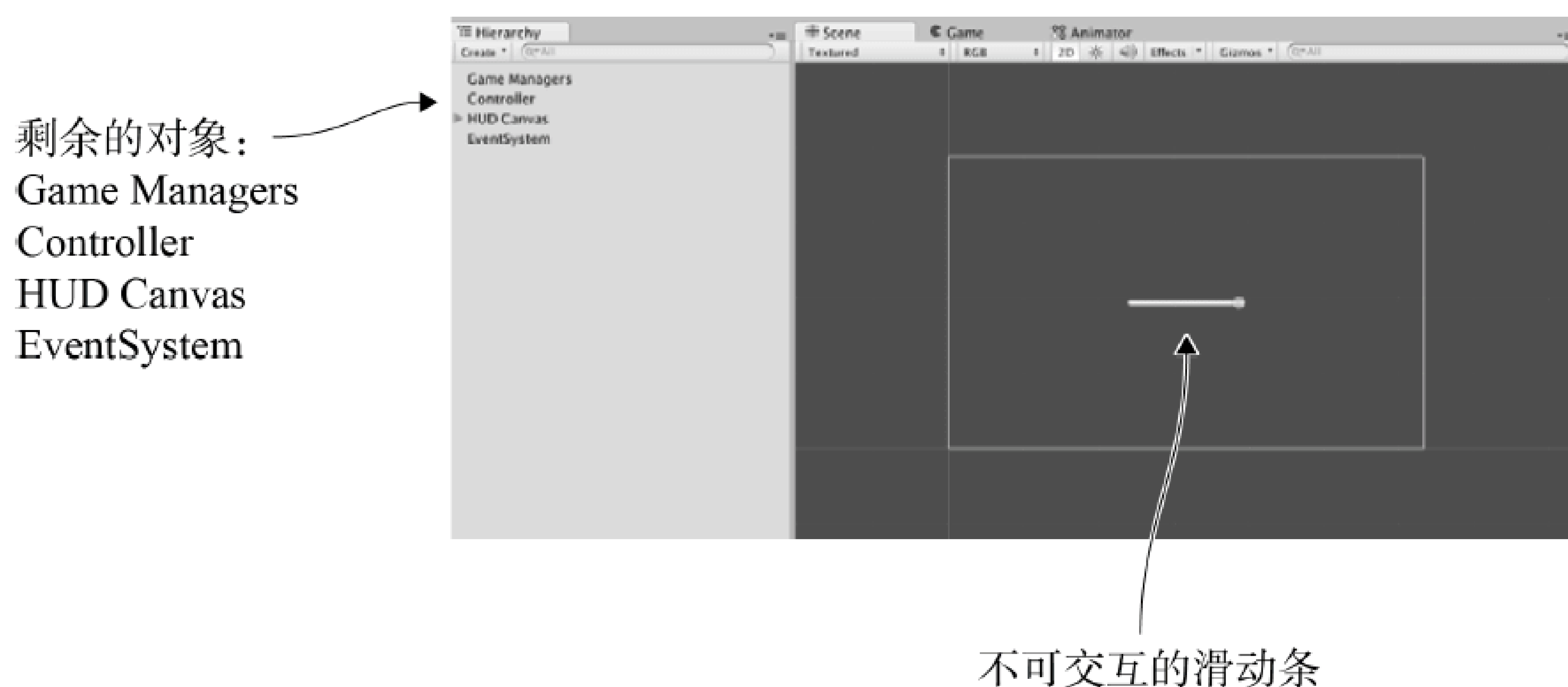


图 12-7　包含所有不必移除的对象的 Startup 场景

代码清单 12.18　新的 StartupController 脚本

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class StartupController : MonoBehaviour {
   [SerializeField] private Slider progressBar;

   void Awake() {
      Messenger<int, int>.AddListener(StartupEvent.MANAGERS_ PROGRESS,
        OnManagersProgress);
      Messenger.AddListener(StartupEvent.MANAGERS_STARTED,
         OnManagersStarted);
   }
   void OnDestroy() {
      Messenger<int, int>.RemoveListener(StartupEvent.MANAGERS_PROGRESS,
         OnManagersProgress);
      Messenger.RemoveListener(StartupEvent.MANAGERS_STARTED,
         OnManagersStarted);
   }

   private void OnManagersProgress(int numReady, int numModules) {
      float progress = (float)numReady / numModules;
      progressBar.value = progress;
   }
```

更新滑动条，显示加载进度

```
    private void OnManagersStarted() {
      Managers.Mission.GoToNext();
    }
  }
```

一旦管理器启动，就加载下一个场景

接下来，将 Slider 对象连接到 Inspector 的槽上。最后要做的准备是将两个场景添加到 Build Settings 中。构建应用是下章的讨论话题，因此现在只需要选择 File | Build Settings，查看并调整场景列表。单击 Add Current 按钮，将当前场景添加到列表中(加载两个场景，对每个场景执行这个操作)。

注意 需要将场景添加到 Build Settings 中，这样可以加载它们。如果不这样做，Unity 将不知道什么场景是可用的。在第 5 章中不需要这样做，因为不需要切换关卡——只是重载当前场景。

现在可以在 Startup 场景中单击 Play，来运行游戏。Game Managers 对象将在两个场景中共享。

警告 因为管理器在 Startup 场景中加载，所以通常需要从 Startup 场景中启动游戏。记住，通常要在单击 Play 之前打开 Startup 场景，但在 Unify 维基上有个脚本，可以在单击 Play 时自动将场景切换为所设置的场景：http://wiki.unity3d.com/index.php/SceneAutoLoader。

提示 默认情况下，加载关卡时，照明系统会重新生成灯光地图，但这只在编辑关卡时有效。运行游戏，在加载关卡时不会生成灯光地图。如第 10 章所述，可以关闭 Lighting 窗口中的 Auto lighting(Window | Lighting)，然后单击按钮，手动生成灯光地图(记住，不要访问所创建的照明文件夹)。

这种结构上的变化可以在不同场景间共享游戏管理器，但依然没有在关卡中设置任何成功/失败条件。

12.2.2 到达出口，完成一个关卡

为了处理关卡的完成，在场景中放置一个对象，让玩家触碰，玩家到达目标时，该对象就通知 MissionManager。这将通知 UI 响应关于关卡完成的消息，因此添加另一个 GameEvent(见代码清单 12.19)。

代码清单 12.19 给 GameEvent.cs 添加关卡完成

```
public static class GameEvent {
  public const string HEALTH_UPDATED = "HEALTH_UPDATED";
  public const string LEVEL_COMPLETE = "LEVEL_COMPLETE";
```

```
}
```

现在，为了跟踪任务目标并广播新事件消息，给 MissionManager 添加一个新方法(见代码清单 12.20)。

代码清单 12.20　MissionManager 中的目标方法

```
...
public void ReachObjective() {
   // could have logic to handle multiple objectives
   Messenger.Broadcast(GameEvent.LEVEL_COMPLETE);
}
...
```

调整 UIController 脚本，以响应该事件(见代码清单 12.21)。

代码清单 12.21　UIController 中的新事件侦听器

```
...
[SerializeField] private Text levelEnding;
...
void Awake() {
   Messenger.AddListener(GameEvent.HEALTH_UPDATED, OnHealthUpdated);
   Messenger.AddListener(GameEvent.LEVEL_COMPLETE, OnLevelComplete);
}
void OnDestroy() {
   Messenger.RemoveListener(GameEvent.HEALTH_UPDATED, OnHealthUpdated);
   Messenger.RemoveListener(GameEvent.LEVEL_COMPLETE, OnLevelComplete);
}
...
void Start() {
   OnHealthUpdated();

   levelEnding.gameObject.SetActive(false);
   popup.gameObject.SetActive(false);
}
...
private void OnLevelComplete() {
   StartCoroutine(CompleteLevel());
}
private IEnumerator CompleteLevel() {
   levelEnding.gameObject.SetActive(true);
   levelEnding.text = "Level Complete!";

   yield return new WaitForSeconds(2);    // 显示消息两秒钟，接着进入下一个关卡

   Managers.Mission.GoToNext();
}
...
```

注意，这个代码清单包含一个对文本标签的引用。打开 Level1 场景，编辑它，创建一个新的 UI 文本对象。这个标签是出现在屏幕中间的关卡完成消息。设置文本的 Width 为 240，Height 为 60，水平和垂直对齐都是居中，Font Size 为 22。在文本区域中输入“Level Complete!”，然后将这个文本对象连接到 UIController 的 levelEnding 引用。

最后，创建一个和玩家接触以完成关卡的对象(图 12-8 显示了这个对象)。这和可收集的物品很类似：它需要一个材质和一个脚本，接下来制作整个预设。

在位置(18, 1, 0)创建一个立方体对象。选择 Box Collider 的 Is Trigger 选项，关闭 Mesh Renderer 的 Cast 和 Receive Shadows，并将对象设置为 Ignore Raycast layer。创建一个新的称为 objective 的材质，将它设置为明亮的绿色，将着色器设置为 Unlit | Color，使得看起来平滑、明亮。

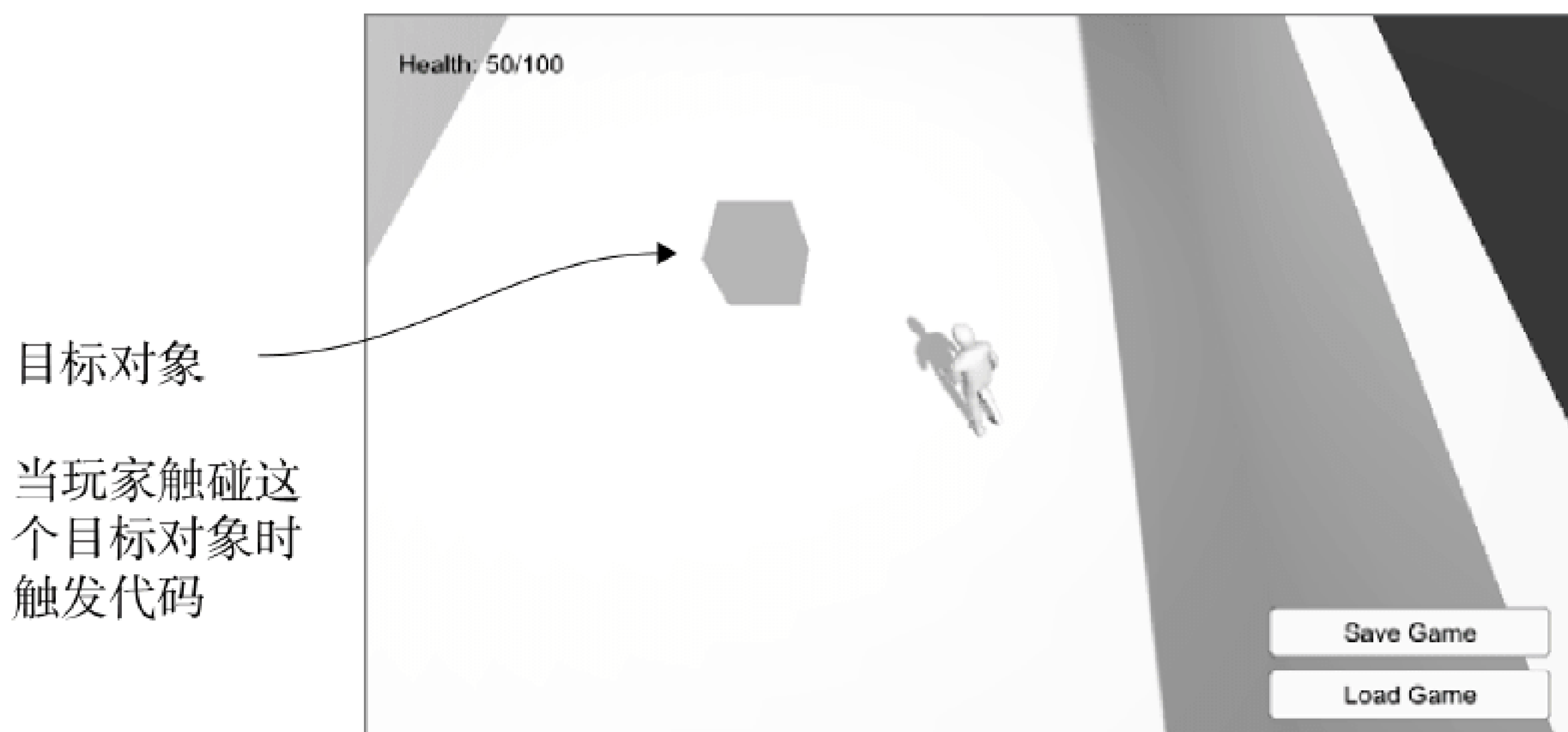


图 12-8　用于被玩家触碰以完成关卡的目标对象

接下来，创建 ObjectiveTrigger 脚本(见代码清单 12.22)，并将其附加到目标对象上。

代码清单 12.22　附加到目标对象上的 ObjectiveTrigger 代码

```
using UnityEngine;
using System.Collections;

public class ObjectiveTrigger : MonoBehaviour {
   void OnTriggerEnter(Collider other) {
      Managers.Mission.ReachObjective();
   }
}
```

调用 MissionManager 中的新目标方法

从 Hierarchy 中将这个对象拖到 Project 视图中，将它变成预设，在未来的关卡中，可以将这个预设放在场景中。现在运行游戏，让角色走向目标。当角色到达目标时，将显示完成的消息。

接下来在失败时显示失败消息。

12.2.3　被敌人捉到时关卡失败

失败条件是指玩家消耗完血量(由于敌人的攻击)。首先添加另一个 GameEvent：

```
public const string LEVEL_FAILED = "LEVEL_FAILED";
```

现在调整 PlayerManager，使玩家血量降为 0 时广播这个消息(见代码清单 12.23)。

代码清单 12.23　从 PlayerManager 广播关卡失败

```
...
public void Startup(NetworkService service) {
   Debug.Log("Player manager starting...");

   _network = service;

   UpdateData(50, 100);      ← 调用更新方法而不是直接设置变量

   status = ManagerStatus.Started;
}

public void UpdateData(int health, int maxHealth) {
   this.health = health;
   this.maxHealth = maxHealth;
}

public void ChangeHealth(int value) {
   health += value;
   if (health > maxHealth) {
      health = maxHealth;
   } else if (health < 0) {
      health = 0; }

   if (health == 0) {
      Messenger.Broadcast(GameEvent.LEVEL_FAILED);
   }
   Messenger.Broadcast(GameEvent.HEALTH_UPDATED);
}

public void Respawn() {      ← 将玩家重置为初始状态
   UpdateData(50, 100);
}
...
```

将一个方法添加到 MissionManager 中，重新启动关卡(见代码清单 12.24)。

代码清单 12.24　可以重新开始当前关卡的 MissionManager

```
...
public void RestartCurrent() {
   string name = "Level" + curLevel;
   Debug.Log("Loading " + name);
   Application.LoadLevel(name);
} ...
```

处理完这些后，将另一个事件侦听器添加到 UIController 中(见代码清单 12.25)。

代码清单 12.25　在 UIController 中响应关卡失败

```
...
Messenger.AddListener(GameEvent.LEVEL_FAILED, OnLevelFailed);
```

```
...
Messenger.RemoveListener(GameEvent.LEVEL_FAILED, OnLevelFailed);
...
private void OnLevelFailed() {
   StartCoroutine(FailLevel());
}
private IEnumerator FailLevel() {
   levelEnding.gameObject.SetActive(true);      ← 重用相同的文本标签，但设置为不同的消息
   levelEnding.text = "Level Failed";

   yield return new WaitForSeconds(2);

   Managers.Player.Respawn();                   ← 在暂停两秒后重新开始当前关卡
   Managers.Mission.RestartCurrent();
}
...
```

运行游戏，让敌人多次射向玩家，最后将出现关卡失败的消息。现在玩家可以完成关卡，或者使关卡失败！在这基础上，游戏必须跟踪玩家的进度。

12.3 处理玩家在游戏过程中的进度

现在各个关卡独立操作，和整个游戏没有任何联系。接下来添加两个处理代码，使游戏的进度更加完整：保存玩家进度，检测游戏(不仅是关卡)何时完成。

12.3.1 保存并加载玩家进度

保存和加载游戏是大多数游戏中重要的一部分。Unity 和 Mono 提供了 I/O 功能来实现该任务。但在开始使用之前，必须在 MissionManager 和 InventoryManager 中添加 UpdateData()。这个方法仅在 PlayerManager 中工作，允许管理器外部的代码在管理器中更新数据。代码清单 12.26 和代码清单 12.27 展示了管理器的变化。

代码清单 12.26 MissionManager 中的 UpdateData()方法

```
...
public void Startup(NetworkService service) {
   Debug.Log("Mission manager starting...");

   _network = service;

   UpdateData(0, 1);                            ← 使用新方法修改这一行

   status = ManagerStatus.Started;
}

public void UpdateData(int curLevel, int maxLevel) {
```

```
    this.curLevel = curLevel;
    this.maxLevel = maxLevel;
}
...
```

代码清单 12.27　InventoryManager 中的 UpdateData()方法

```
...
public void Startup(NetworkService service) {
    Debug.Log("Inventory manager starting...");

    _network = service;

    UpdateData(new Dictionary<string, int>());    ← 初始化一个空列表

    status = ManagerStatus.Started;
}

public void UpdateData(Dictionary<string, int> items) {
    _items = items;
}
                                                  为了保存游戏数据，需
public Dictionary<string, int> GetData() {    ← 要 getter 方法
    return _items;
}
...
```

现在不同的管理器都有了 UpdateData()方法，数据可以在一个新代码模块中保存。保存数据，涉及一个被称为序列化数据的过程。

定义　*序列化(serialize)意味着将一批数据编码为可以保存的形式。*

接下来将游戏保存为二进制数据，但注意，C#也完全能胜任保存文本文件的任务。例如，第 10 章中使用的 JSON 字符串就是把数据序列化为文本。之前的章节使用的是 PlayerPrefs，但本项目保存为本地文件(PlayerPrefs 仅用于保存少量的值，例如设置，而不是整个游戏)。创建 DataManager 脚本(见代码清单 12.28)。

警告　*不能在网页游戏中访问文件系统。这是 Web 浏览器的一个安全特性。为了保存网页游戏的数据，需要编写一个插件，如下一章所述，或将数据发送到服务器。*

代码清单 12.28　用于 DataManager 的新脚本

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

public class DataManager : MonoBehaviour, IGameManager {
    public ManagerStatus status {get; private set;}
```

```
    private string _filename;

    private NetworkService _network;

    public void Startup(NetworkService service) {
      Debug.Log("Data manager starting...");

      _network = service;

      _filename = Path.Combine(
          Application.persistentDataPath, "game.dat");   ← 构建 game.dat 文件的完整路径

      status = ManagerStatus.Started;
    }

    public void SaveGameState() {
      Dictionary<string, object> gamestate =
        new Dictionary<string,object>();                 ← 被序列化的字典
      gamestate.Add("inventory", Managers.Inventory.GetData());
      gamestate.Add("health", Managers.Player.health);
      gamestate.Add("maxHealth", Managers.Player.maxHealth);
      gamestate.Add("curLevel", Managers.Mission.curLevel);
      gamestate.Add("maxLevel", Managers.Mission.maxLevel);

      FileStream stream = File.Create(_filename);       ← 在文件路径中创建一个文件
      BinaryFormatter formatter = new BinaryFormatter();
      formatter.Serialize(stream, gamestate);           ← 序列化字典作为所建文件的内容
      stream.Close();
    }

    public void LoadGameState() {
      if (!File.Exists(_filename)) {                    ← 只有当文件存在时才继续加载
        Debug.Log("No saved game");
        return;
      }

      Dictionary<string, object> gamestate;             ← 用于放置所加载数据的字典

      BinaryFormatter formatter = new BinaryFormatter();
      FileStream stream = File.Open(_filename, FileMode.Open);
      gamestate = formatter.Deserialize(stream) as Dictionary<string,
      object>;
      stream.Close();

      Managers.Inventory.UpdateData((Dictionary<string,
     int>)gamestate["inventory"]);                      ← 使用反序列化的数据更新管理器
      Managers.Player.UpdateData((int)gamestate["health"],
     (int)gamestate["maxHealth"]);
      Managers.Mission.UpdateData((int)gamestate["curLevel"],
     (int)gamestate["maxLevel"]);
      Managers.Mission.RestartCurrent();
    }
  }
```

在 Startup()期间，使用 Application.persistentDataPath 构建完整的文件路径，这是

Unity 提供的用于保存数据的位置。文件的精确位置在不同的平台上不同，但 Unity 将它抽象于这个静态变量的背后(这个路径包括 Player Settings 中的 Company Name 和 Product Name，因此，如果有需要，就可以调整它)。File.Create()方法会创建一个二进制文件；如果想创建一个文本文件，就调用 File.CreateText()方法。

警告　当构建文件路径时，路径分隔符在不同的计算机平台上是不同的。C#中通过 Path.DirectorySeparatorChar 来处理路径分隔符在不同平台上的差异性。

打开 Startup 场景，找到 Game Managers。将 DataManager 脚本组件添加到 Game Managers 对象上，接着将新的管理器添加到 Managers 脚本中(见代码清单 12.29)。

代码清单 12.29　将 DataManager 添加给 Managers.cs

```
...
[RequireComponent(typeof(DataManager))]
...
public static DataManager Data {get; private set;}
...
void Awake() {
  DontDestroyOnLoad(gameObject);

  Data = GetComponent<DataManager>();
  Player = GetComponent<PlayerManager>();
  Inventory = GetComponent<InventoryManager>();
  Mission = GetComponent<MissionManager>();

  _startSequence = new List<IGameManager>();    ← 管理器以这个顺序启动
  _startSequence.Add(Player);
  _startSequence.Add(Inventory);
  _startSequence.Add(Mission);
  _startSequence.Add(Data);

  StartCoroutine(StartupManagers());
}
...
```

警告　因为 DataManager 使用其他管理器(以更新它们)，所以应该确保其他管理器在启动序列中出现在 DataManager 之前。

最后，在 Level1 中添加按钮，以使用 DataManager 中的函数(图 12-9 展示了这些按钮)。创建两个按钮，使它们的父节点为 HUD Canvas(而不是 Inventory 弹出窗口)。将它们命名(设置附加的文本对象)为 Save Game 和 Load Game，将 Anchor 设置在右下角，将这两个按钮定位在(−100, 65)和(−100, 30)。

这些按钮将连接到 UIController 中的函数，因此编写这些方法(见代码清单 12.30)。

代码清单 12.30　UIController 中的 Save 和 Load 方法

```
...
public void SaveGame() {
   Managers.Data.SaveGameState();
}

public void LoadGame() {
   Managers.Data.LoadGameState();
}
...
```

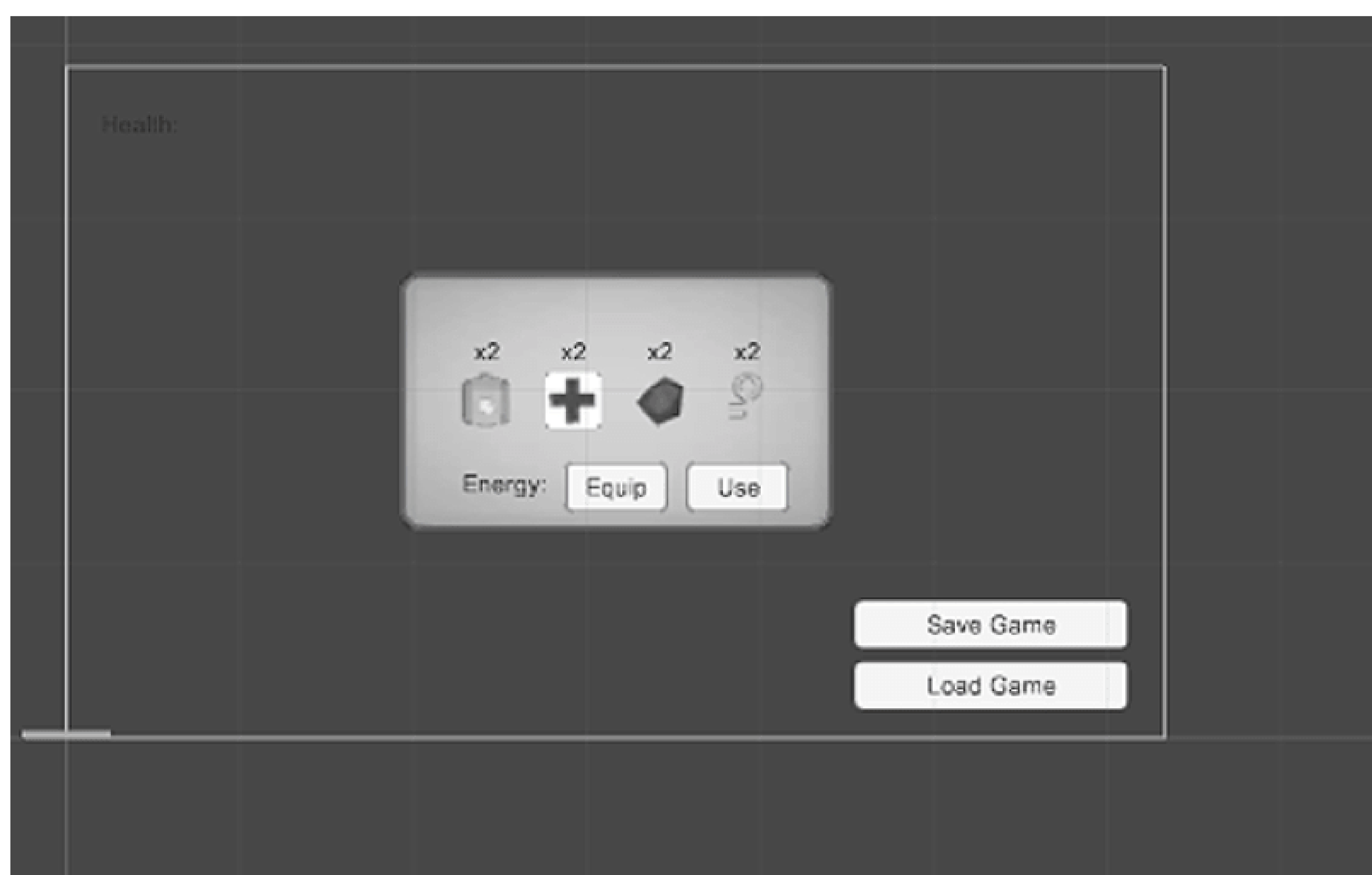


图 12-9　屏幕右下角的 Save 和 Load 按钮

将这些函数连接到按钮的 OnClick 侦听器(在 OnClick 设置中添加一个代码清单，将其拖入 UIController 对象，从菜单中选择函数)。现在运行游戏，捡起一些物品，使用血量包增加血量，接着保存游戏。重启游戏，检查仓库，以验证它是否为空。单击 Load，现在保存游戏时，就有了血量和物品的信息。

12.3.2　完成三个关卡，游戏通关

保存玩家进度暗示着，这个游戏可以有多个关卡，而不仅仅是当前测试的一个关卡。为了正确处理多个关卡，游戏不仅必须检查一个关卡的完成进度，还要检查整个游戏的完成进度。首先添加另一个 GameEvent：

```
public const string GAME_COMPLETE = "GAME_COMPLETE";
```

现在修改 MissionManager，在最后一个关卡(见代码清单 12.31)之后广播消息。

代码清单 12.31　在 MissionManager 中广播游戏完成

```
...
public void GoToNext() {
   ...
   } else {
```

```
            Debug.Log("Last level");
            Messenger.Broadcast(GameEvent.GAME_COMPLETE);
        }
    }
```

在 UIController 中响应该消息(见代码清单 12.32)。

代码清单 12.32　将事件侦听器添加到 UIController 中

```
...
Messenger.AddListener(GameEvent.GAME_COMPLETE, OnGameComplete);
...
Messenger.RemoveListener(GameEvent.GAME_COMPLETE, OnGameComplete);
...
private void OnGameComplete() {
    levelEnding.gameObject.SetActive(true);
    levelEnding.text = "You Finished the Game!";
}
...
```

尝试完成关卡，并观察会发生什么情况：将玩家移动到关卡目标，完成关卡。首先会显示 Level Complete 消息，但在几秒后，该消息会改为游戏完成消息。

添加更多关卡

此时，可以添加任意数量的额外关卡，而 MissionManager 将侦听最后的关卡。本章最后需要将更多的关卡添加到项目中，以演示多关卡的游戏进度。

将 Level1 场景文件复制两次(Unity 应该自动递增数字为 Level2 和 Level3)，将新关卡添加到 Build Settings 中(以便可以在游戏运行过程中加载，记得要生成照明效果)。修改每个场景，以找出不同关卡的区别。随意重新排列场景，但必须保留一些必需的游戏元素：标记为 Player 的玩家对象、设置为 Ground 层的地板对象和目标对象、Controller、HUD Canvas 和 EventSystem。

还需要调整 MissionManager 来加载新关卡。将调用 UpdateData(0, 1)改为调用 UpdateData(0, 3)，从而将 maxLevel 更改为 3。

现在运行游戏，最初从 Level1 开始，到达关卡目标后进入下一关卡！也可以在后续的关卡中保存，看看游戏从哪个进度恢复。

练习：将音频集成到完整的游戏中

第 11 章介绍了如何在 Unity 中实现音频。在此不解释如何将音频集成到本章项目中，但此时读者应该知道实现方式。鼓励读者将前面章节的音频功能集成到本章项目，提高自己的技能。提示：修改用于开关音频设置弹出窗口的键，使它不和仓库弹出窗口冲突。

现在知道了如何创建带有多个关卡的完整游戏。显然，下一个任务是最后一章的内容：将游戏交到玩家手中。

12.4　小结

- Unity 简化了从不同类型的游戏项目中重用资源和代码的过程。
- 射线发射的另一个主要用途是判定玩家单击了场景的哪个位置。
- Unity 为加载关卡和在关卡间持久化对象提供了一些简单的方法。
- 通过响应游戏中不同的事件推进关卡进度。
- 可以使用 C#提供的 I/O 方法在 Application.persistentDataPath 中存储数据。

第 13 章

将游戏部署到玩家的设备

本章涵盖：

- 为不同的平台构建应用包
- 设定发布设置，例如应用图标或名称
- 为了 Web 游戏与网页交互
- 在移动平台上为应用开发插件

只要通读本书，就能学会如何在 Unity 中编写不同的游戏，但还缺失重要的最后一步：将游戏部署给玩家。游戏在 Unity 编辑器外运行之前，除了开发者，没有其他人对它有兴趣。Unity 在这最后一步有一些优势，可以为多种不同的游戏平台构建应用。本章将讲解如何为这些不同平台构建游戏。

为平台构建游戏，意味着生成在该平台上运行的应用包。在每个平台(Windows、iOS 等)上构建应用的具体形式也不同，但一旦生成了可运行程序，应用包就可以在 Unity 外运行，并分发给玩家。单一的 Unity 项目可以部署到任何平台，不需要重做。

“构建一次，到处部署”的能力适用于游戏的大多数

主要特性，但不是全部。在 Unity 中编写的所有代码(例如，本书完成的几乎所有代码)，大约 95%是与平台无关的，可以很好地在所有平台上工作。但一些特殊的任务对于不同平台是不同的，因此接下来讨论与平台相关方面的开发。

Unity 可以为下列平台构建应用：

- Windows PC
- Mac OS X
- Linux
- WebGL
- iOS
- Android
- Windows Phone
- Tizen
- Oculus Rift
- Steam VR/Vive
- Daydream
- HoloLens

此外，与平台的拥有者达成协议，Unity 可以为下列平台构建应用：

- XBox One
- PlayStation 4
- PS Vita
- Switch
- 3DS

完整的列表很长，远远多于其他大多数游戏开发工具支持的平台。本章将关注其中的前 6 个平台，因为那些平台是大多数 Unity 探索者主要感兴趣的，但是要记住可用的平台数量。

为了查看所有这些平台，打开 Build Settings 窗口。前面章节使用该窗口添加被加载的场景，为了访问这个窗口，选择 File | Build Settings。第 12 章只讨论了列表顶部的几个选项，现在应该关注底部的按钮(如图 13-1 所示)。注意平台列表占用了很多空间，Unity 图标指明了当前激活的平台。在这个列表中选择平台，单击 Switch Platform 按钮。

注意　在安装 Unity 时，安装程序要求用户指定想要哪个导出模块，可以只构建所选的模块。如果稍后要安装最初没有选择的模块，在 Build Settings 窗口中有一个添加它的按钮。

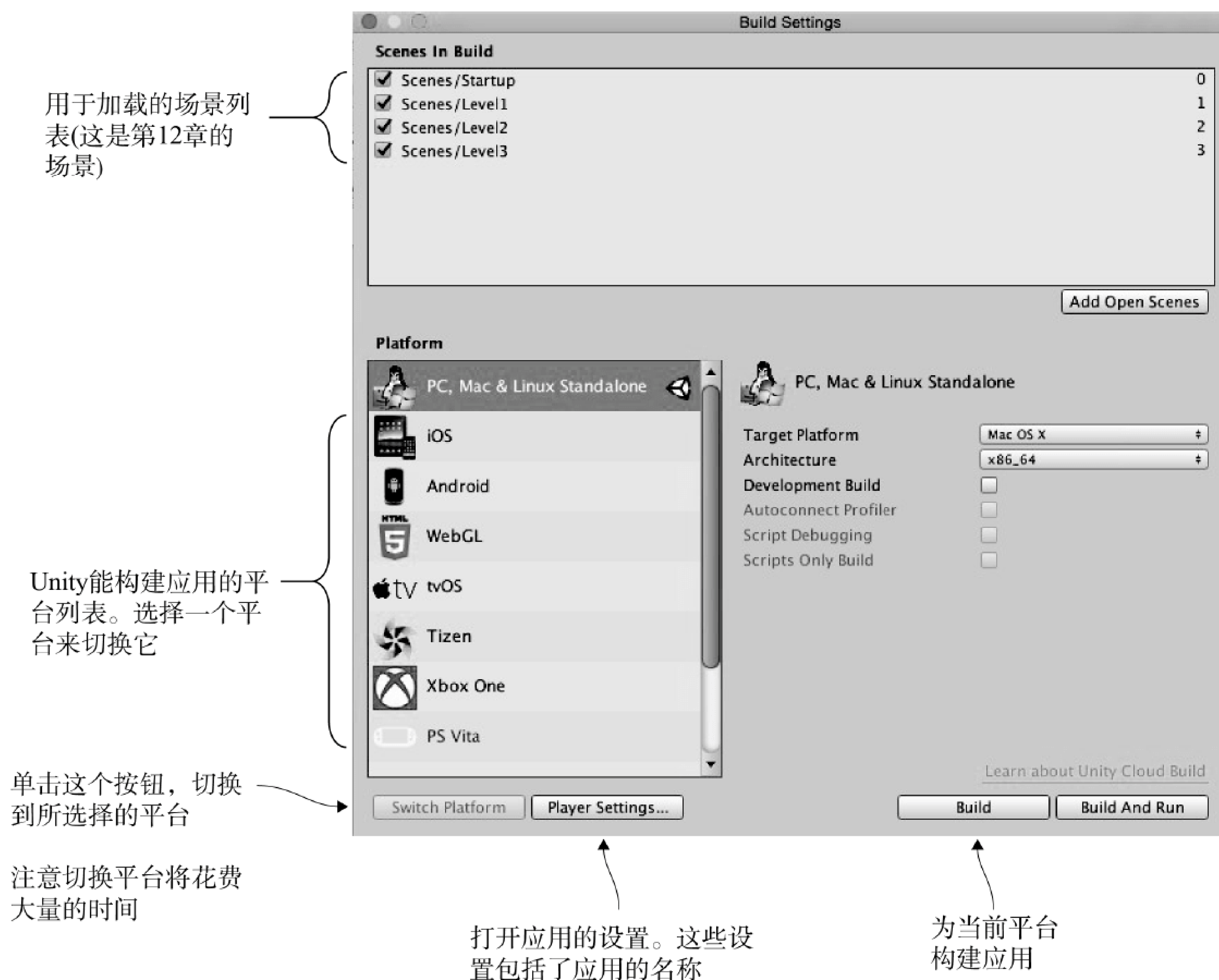


图 13-1　Build Settings 窗口

警告　在大型项目中，切换平台通常需要花费很长的时间，请确认自己愿意等待。这是因为 Unity 以适合每个平台的方式重新压缩了所有资源(例如，贴图)。

窗口底部也有 Player Settings 和 Build 按钮。单击 Player Settings，在 Inspector 中查看应用的设置，例如名称和图标。单击 Build，启动构建过程。

提示　Build And Run 的作用与 Build 一样，但它会自动运行并构建应用游戏。通常希望手动运行，因此很少使用 Build And Run。

单击 Build 时，首先会弹出一个文件选择器，以便告诉 Unity 在哪里生成应用包。一旦选择了文件位置，构建过程将开始。Unity 为当前激活的平台创建可运行的应用包，接下来介绍大多数主流平台的构建过程：桌面、Web、移动。

13.1　构建用于桌面的应用包：Windows、Mac 和 Linux

当首次学习构建 Unity 游戏时，最简单的起点是将其部署到台式机——Windows PC、

Mac OS X 或 Linux。Unity 运行在台式机上，意味着为所使用的计算机构建应用。

注意 打开本节中要使用的任何项目，严格来说，任何 Unity 项目都可以。强烈建议在每节中使用不同的项目，以理解 Unity 可以将任何项目编译到任何平台这个事实！

13.1.1　构建应用

首先选择 File | Build Settings，打开 Build Settings 窗口。默认情况下，当前平台设置为 PC、Mac 和 Linux，但如果当前平台不是上述平台，则可以从列表中选择正确的平台，单击 Switch Platform。

在窗口右边有一个 Target Platform 菜单。这个菜单允许在 Windows PC、Mac OS X 和 Linux 之间选择。左边的列表把这 3 个平台当成一个平台，但这三个平台的差异很大，因此要选择正确的那个。

一旦选择了桌面平台，就单击 Build，弹出一个文件对话框，允许选择应用的构建位置。之后开始构建过程，对于大型项目，这可能会需要一段时间，但对于前面制作的小型演示游戏，构建过程应该会执行得很快。

自定义构建后脚本

尽管基本的构建过程适用于大多数情形，但每次构建游戏时，可以执行一些步骤(例如，将帮助文件移到与应用相同的目录中)。在脚本中编写程序，在构建过程完成后执行该脚本，就可以使这些任务自动化。

首先，在Project视图中创建文件夹，将之命名为Editor。任何影响Unity编辑器(包括构建过程)的脚本必须放在Editor文件夹中，创建一个新脚本，命名为TestPostBuild。编写如下代码清单：

```
using UnityEngine;
using UnityEditor;
using UnityEditor.Callbacks;

public static class TestPostBuild {

    [PostProcessBuild]
    public static void OnPostprocessBuild(BuildTarget target, string
pathToBuiltProject) {
        Debug.Log("build location: " + pathToBuiltProject);
    }
}
```

指令[PostProcessBuild] 告诉脚本，在构建完之后立刻运行这个方法。这个方法将接收应用构建的位置，这样就可以对那个路径使用 C#提供的各种文件系统命令。

应用将出现在选定的位置，双击并运行它，就像其他程序一样。这很简单！构建应用很迅速，但该过程可以以各种方式自定义。下面看看如何调整构建过程。

在 Windows 上使用 Alt+F4 组合键或在 Mac 上使用 Cmd+Q 组合键，退出全屏游戏。为了结束游戏，应该有一个调用 Application.Quit()方法的按钮。

13.1.2　调整 Player Settings：设置游戏的名称和图标

回到 Build Settings 窗口，但这次单击 Player Settings 而不是 Build。在 Inspector 中会显示一个很长的设置列表(如图 13-2 所示)，这些设置控制了应用构建过程的某些方面。

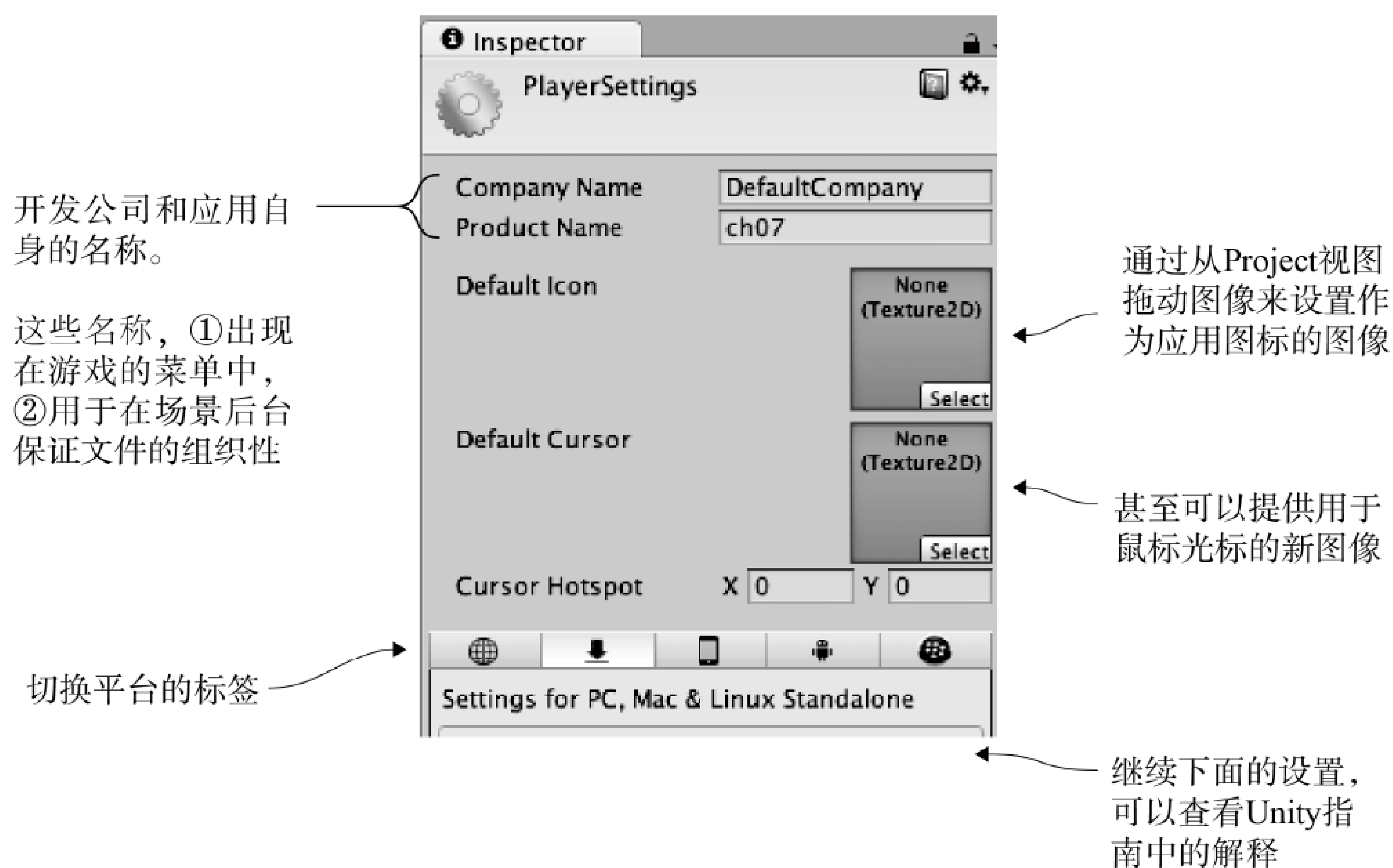


图 13-2　显示在 Inspector 中的玩家设置

因为有大量设置，所以可能需要在 Unity 指南中查看它们的使用方法，相关的文档页面在：http://docs.unity3d.com/Manual/class- PlayerSettings.html。

顶部的前三个设置最容易理解：Company Name、Product Name 和 Default Icon。为前两个设置输入值。Company Name 是开发工作室的名称，而 Product Name 是这个产品的名称。接着从 Project 视图(如果需要，导入一个图像到项目中)中拖动图像，以设置图像为图标，当应用构建完毕后，这个图像将作为应用的图标。

支持虚拟现实(Virtual Reality,VR)

Unity 为 Oculus 和 Vive 等 VR 硬件提供了内置的特殊支持，但在构建应用程序时，VR 在技术上并不是一个独立的平台。相反，支持 VR 是可以在相关的构建平台上切换的设置，比如桌面 VR 或者 Android。进入 Player Settings 中的正确选项卡(注意平台按钮在顶部的部分)，然后展开 Other Settings 部分。其中有一个切换选项 Virtual Reality Supported，开发 VR 时打开它，如图 13-3 所示。

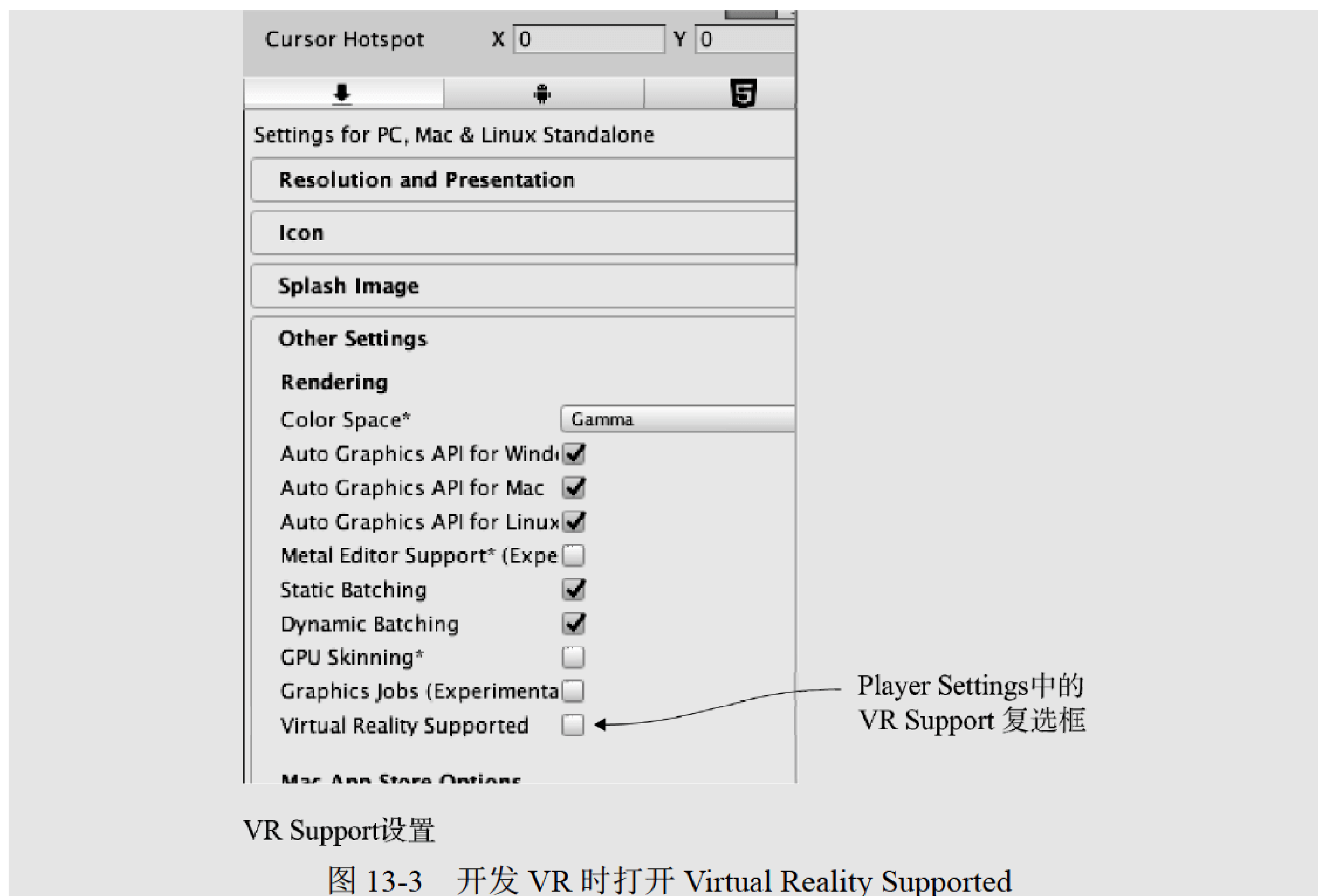


图 13-3　开发 VR 时打开 Virtual Reality Supported

自定义应用的图标和名称对于已完成游戏的外观很重要。自定义构建应用行为的另一个重要方式是使用平台依赖的代码。

13.1.3　平台依赖的编译

默认情况下，我们编写的所有代码在所有平台上都以相同的方式运行。但 Unity 提供了一些编译器指令(称为平台定义)，让不同代码运行在不同平台上。在 Unity 指南的如下页面中可以找到完整的平台定义列表：http://docs.unity3d.com/Manual/PlatformDependentCompilation.html。

如页面所示，Unity 对每个平台都提供了支持的指令。通常大部分代码不一定包含在平台指令中，但偶尔小部分代码需要在不同平台上执行不同的处理。一些代码程序集仅在一个平台上存在，因此需要让平台编译器指令包含那些命令。代码清单 13.1 展示了如何编写这样的代码。

代码清单 13.1　PlatformTest 脚本展示了如何编写平台依赖的代码

```
using UnityEngine;
using System.Collections;

public class PlatformTest : MonoBehaviour {
   void OnGUI() {
#if UNITY_EDITOR
      GUI.Label(new Rect(10, 10, 200, 20), "Running in Editor");
```

这部分只运行在编辑器中

```
#elif UNITY_STANDALONE                                    ← 仅在桌面/单机应用中
     GUI.Label(new Rect(10, 10, 200, 20), "Running on Desktop");
#else
     GUI.Label(new Rect(10, 10, 200, 20), "Running on other platform");
#endif
   }
}
```

创建一个称为 PlatformTest 的脚本，编写上述代码清单中的代码。将这个脚本附加到场景的对象上(对任何对象都可以做这个测试)，一个小图像将出现在屏幕左上角。在 Unity 编辑器中运行游戏时，消息将显示“Running in the Editor”。但如果构建游戏，并运行构建好的应用，消息就变成“Running On Desktop”。在不同情况下将运行不同代码！

对于这个测试，将所有桌面平台视为一个平台定义，也可以如文档页面所示，对于 Windows、Mac 和 Linux 有可用的独立平台定义。实际上，Unity 支持的所有平台都有平台定义，因此可以在它们上面运行不同的代码。接下来解释下一个重要的平台：Web。

质量设置

构建的应用也受 Edit 菜单下的项目设置(project settings)影响。特别是可以在此调整最终应用的视觉质量。单击 Edit 菜单中的 Project Settings，从下拉菜单中选择 Quality。

Quality 设置显示在 Inspector 中，而最重要的设置是顶部的一系列复选框。Unity 可以发布的不同目标平台以图标的形式在顶部列出，而可能的质量设置也在旁边列出来。选中可用于平台的质量设置复选框，要使用的复选框突出显示为绿色。大多数情况下，这些设置默认是 Fastest(质量最差)，但如果质量太差，也可以改为 Fantastic 质量。如果单击平台的列下面的下拉箭头，将出现一个弹出菜单。

UI 同时有复选框和默认菜单，看起来有点多余，但确实需要它们。不同平台通常有不同的图形特性，因此 Unity 允许为不同的构建目标设置不同的质量级别(例如，在桌面平台使用最好质量，而在移动平台使用较差质量)，如图 13-4 所示。

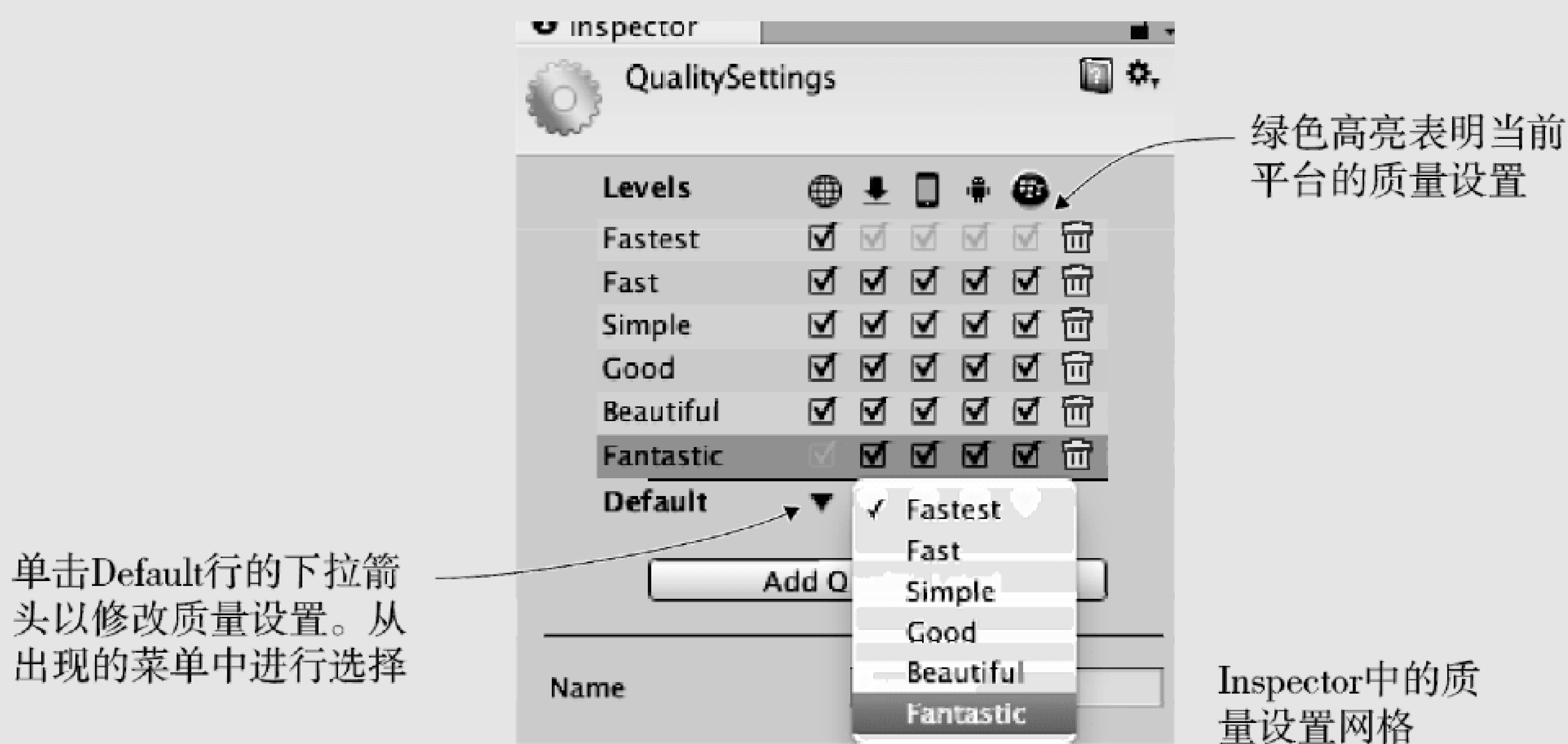


图 13-4　Unity 为不同的构建目标设置不同的质量级别

13.2　为 Web 构建游戏

桌面平台是构建的最基础目标，而 Unity 游戏的另一个重要平台是部署到 Web。这意味着游戏可以在 Web 浏览器内运行，可以通过互联网进行。

13.2.1　Unity Player 和 HTML5/ WebGL

以前，Unity 需要以自定义浏览器插件运行的方式部署 Web。这在很长一段时间内都是必须的，因为 3D 图形并未内置于 Web 浏览器中。然而在随后几年，出现了称为 WebGL 的 Web 3D 图形标准。技术上而言，WebGL 是和 HTML5 分离的，尽管这两个术语有关系且通常可以互换。

Unity5 将 WebGL 添加到了构建窗口的平台列表中，几个版本之后，浏览器插件被删除，使 WebGL 成为进行 Web 构建的唯一途径。在某种程度上，Unity Web 构建的改变受 Unity(公司)的战略决策驱动。这些改变也由浏览器制作商推动而驱动，这些制造商抛弃自定义插件，把 HTML5/WebGL 作为 Web 应用交互的方法，包括游戏。

13.2.2　构建嵌入网页的游戏

打开一个不同的项目(同样，这是为了强调任何项目都可以工作)，打开 Build Settings 窗口。将平台切换到 WebGL，单击 Build 按钮。这将出现一个文件选择器，为这个应用输入名称 WebTest，如果需要，可将其改为一个可靠的位置(即不在 Unity 项目内的位置)。

构建过程现在创建一个包含 index.html 网页的文件夹，再为游戏的所有代码和其他资源创建子文件夹。打开这个 Web 页面，游戏应该嵌入在空页面的中间。

这个页面并没有什么特殊之处，它只是测试游戏的一个例子。可以自定义页面的代码，甚至提供自己的 Web 页面(稍后介绍)。最重要的自定义是允许 Unity 和浏览器间通信，参见下一节。

13.2.3　与浏览器中的 JavaScript 通信

UnityWeb 游戏可以和浏览器(更精确地说是运行在浏览器上的 JavaScript)通信，而这些消息既可以从 Unity 发送到浏览器，也可以从浏览器发送到 Unity。要发送消息到浏览器，可以把 JavaScript 代码编写到代码库中，Unity 有一些特殊的命令用于使用该库中的功能。

对于来自浏览器的消息，浏览器中的 JavaScript 通过名称标识一个对象，然后 Unity 将消息传递给场景中指定的对象。因此场景中必须有一个用于从浏览器接收通信的对象。

为了演示这些任务，在 Unity 中创建一个称为 WebTestObject 的新脚本。同时在激活的场景中创建一个称为 JSListener 的空对象(场景中的对象必须使用准确的名称，因为代码清单 13.4 中的 JavaScript 代码使用了该名称)。将新脚本附加到 JSListener 对象上，接着编写代码清单 13.2 中的代码。

代码清单 13.2　用于测试与浏览器通信的 WebTestObject 脚本

```
using UnityEngine;
using System.Runtime.InteropServices;

public class WebTestObject : MonoBehaviour {
  private string _message;
                                                    从 JS 库中导入函数
  [DllImport("__Internal")]           <-----
  private static extern void ShowAlert(string msg);

  void Start() {
    _message = "No message yet";
  }
   void Start() {
     _message = "No message yet";
   }
                                                    当鼠标单击时，调用
  void Update() {                                   导入的函数
    if (Input.GetMouseButtonDown(0)) {  <-----
       ShowAlert("Hello out there!");
    }
  }
                                                    在屏幕左上角
  void OnGUI() {                                    显示消息
    GUI.Label(new Rect(10, 10, 200, 20), _message);  <-----
  }
                                                    被浏览器调用
  public void RespondToBrowser(string message) {  <----- 的函数
    _message = message;
  }
}
```

主要的新代码是 DLLImport 命令。它从 JavaScript 库中导入一个函数，以便在 C# 代码中使用。这显然意味着有一个 JavaScript 库，所以接下来编写这个库。首先创建包含它的特殊文件夹：创建一个名为 Plugins 的文件夹，在其中创建一个名为 WebGL 的文件夹。现在，将一个名为 WebTest 的文件放在 WebGL 文件夹中，该文件的扩展名为 jslib(即 WebTest.jslib)。最简单的方法是在 Unity 之外创建一个文本文件，重命名它，然后将该文件拖进来。Unity 将这个文件识别为 JavaScript 库，所以在其中编写如下代码。(见代码清单 13.3)

代码清单 13.3　JavaScript 库 WebTest

```
var TestLib = {
```

```
    ShowAlert: function(msg) {
      window.alert(Pointer_stringify(msg));
    },
  }
  mergeInto(LibraryManager.library, TestLib);
```

从 C#中导入并调用的函数

jslib 文件包括一个包含函数的 JavaScript 对象和将定制对象合并到 Unity 库管理器中的命令。注意，除了标准的 JavaScript 命令之外，编写的函数还包括 Pointer_stringify()。从 Unity 传递字符串时，它会变成一个数字标识符，因此 Unity 提供了该函数，来查找它指向的字符串。

现在，再一次构建 Web，以使用新代码。单击 Web 页面的 Unity 游戏部分时，Unity 中的 WebTestObject 会调用 JavaScript 代码中的函数，尝试单击几次，将看到浏览器中的提示框。

注意 Unity 的 Application.ExternalEval()方法运行浏览器中的代码，ExternalEval()运行任意的 JavaScript 片段，而不是调用已定义的函数。这个方法已废弃，应该避免使用，但有时使用是很简单的，比如使用如下代码重新加载页面：

```
Application.ExternalEval("location.reload();");
```

前面在网页上测试了从 Unity 游戏到 JavaScript 的通信，Web 页面也可以将消息发送到 Unity 中，下面完成这个操作。这需要页面上的新代码和按钮，幸运的是，Unity 提供了定制 Web 页面的简单方法。具体来说，Unity 在构建到 WebGL 时，会填充 Web 页面模板，可以选择定制模板，而不是默认模板。

默认模板可以在 Unity 安装文件夹下找到，在 Windows 上该文件夹是 Editor\Data\PlaybackEngines\WebGLSupport\BuildTools\WebGLTemplates，在 Mac 上，该文件夹是/PlaybackEngines/WebGLSupport/BuildTools/WebGLTemplates。在文本编辑器中打开一个模板页面，该模板主要是标准的 HTML 和 JavaScript，加上一些特殊的标签，Unity 会用生成的信息替换这些标签。尽管最好不要仅使用 Unity 的内置模板，但它们(尤其是最小的模板)为构建自己的模板打下了良好的基础。后面将把最小模板网页复制到自己创建的自定义模板中。

在 Unity 中，创建一个名为 WebGLTemplates 的文件夹，在其中保存定制模板。现在在该文件夹中创建一个名为 WebTest 的文件夹，用于保存新模板。把 index.html 文件放在这里(可以在最小模板的网页中复制)，在文本编辑器中打开它，并在其中编写如下代码。(见代码清单 13.4)

代码清单 13.4　支持浏览器-Unity 通信的 WebGL 模板

```
<!doctype html>
<html lang="en-us">
```

```
<head>
<title>Unity WebGL Player | %UNITY_WEB_NAME%</title>
<style>
body { background-color: #333; }
</style>

<script src="%UNITY_WEBGL_LOADER_URL%"></script>
<script>
var gameInstance = UnityLoader.instantiate("gameContainer", "%UNITY_WEBGL_
      BUILD_URL%");

function SendToUnity() {
   gameInstance.SendMessage("JSListener",
      "RespondToBrowser", "Hello from the browser!");

}
</script>
</head>

<body>
<div id="gameContainer" style="width: %UNITY_WIDTH%px; height: %UNITY_
   HEIGHT%px; margin: auto"></div>
<br><input type="button" value="Send" onclick="SendToUnity();" />
</body>
</html>
```

使页面变成黑色，而不是白色

SendMessage()指向 Unity 中指定的对象

调用 JavaScript 函数的按钮

如果复制最小模板，代码清单 13.4 就只是添加了几行代码。两个重要的新代码是标题脚本中的函数和底部添加的输入按钮，添加的样式改变了页面的颜色，这样更容易看到嵌入的游戏。按钮的 HTML 标记链接到一个 JavaScript 函数，该函数在 Unity 实例上调用 SendMessage()。这个方法在 Unity 中调用一个命名对象的函数，第一个参数是对象的名称，第二个参数是方法的名称，第三个参数是调用方法时传入的字符串。

创建了自定义模板后，仍然需要告诉 Unity 使用这个模板而不是默认模板。再次打开 Player Settings (记住，单击 Build Settings 窗口中的 Player Settings)，在 Web 设置中找到 WebGL 模板(如图 13-5 所示)。当前选择了 Default，但是 WebTest(前面创建的模板文件夹)也在列表中，单击它代替 Default。

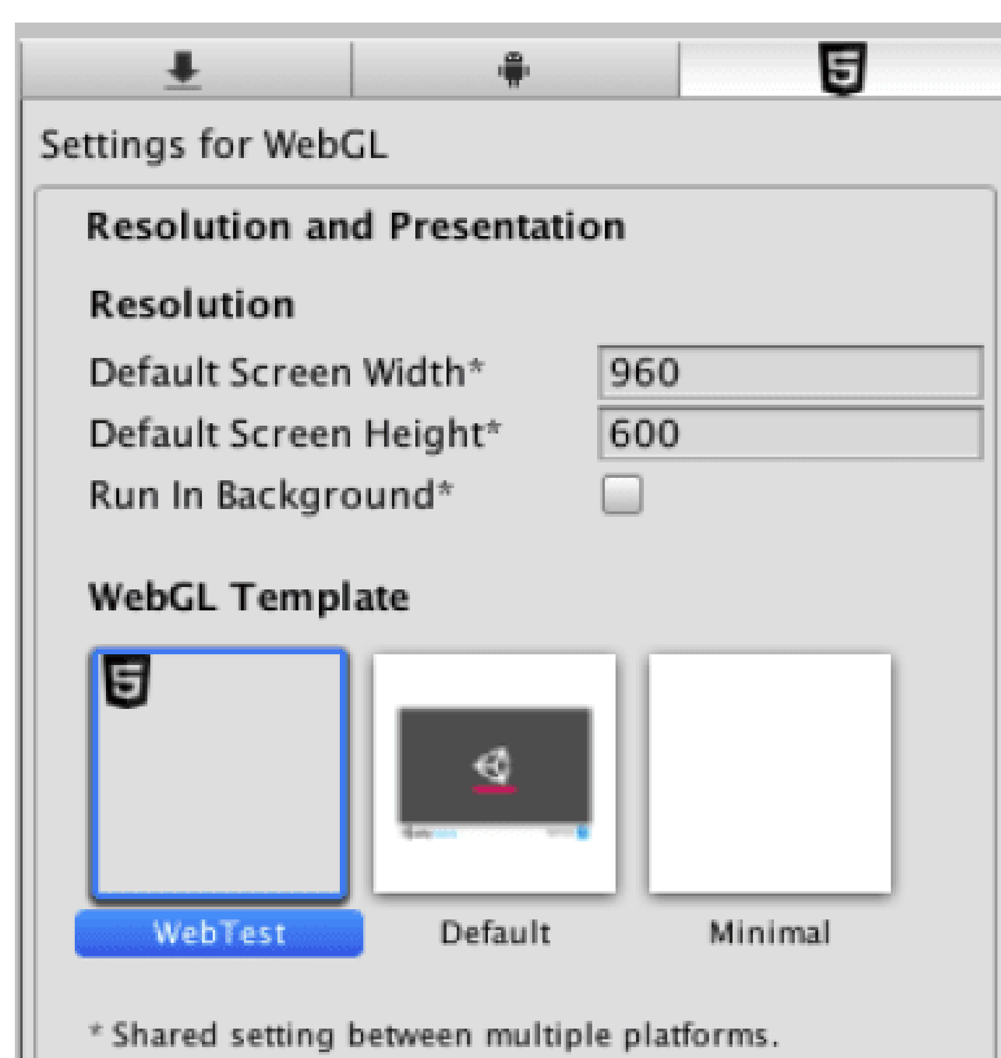


图 13-5　WebGL Template 设置

选中自定义模板后，再次构建到 WebGL。打开生成的 Web 页面，这次在页面底部有一个按钮。单击按钮，Unity 中就会显示更改的消息!

以上介绍了与浏览器通信的 Web 构建，另外还将讨论构建应用的一个平台(或者说，一系列平台)：移动应用。

13.3　构建移动应用的平台：iOS 和 Android

移动应用是 Unity 的另一个重要的构建目标。我们的直观印象(没有科学统计)是，使用 Unity 创建的商业游戏中大部分是移动游戏。

定义　移动指一种手持计算设备。最开始指的是智能手机，但现在包括了平板电脑。两个使用最广的移动计算平台是 iOS(来自苹果)和 Android(来自 Google)。

构建移动应用的设置过程比桌面或 Web 构建更复杂，因此这是一个可选的部分——读者只能阅读这部分内容，实际上并没有执行这些步骤。本章依然会介绍这部分内容，假如读者工作在移动平台上，就应该购买一个用于 iOS 的开发者许可，并为 Android 安装开发工具。

警告　移动设备正在经历巨大的革新，所以具体的构建过程可能会和现在阅读的内容略有不同。高级概念可能依然准确，但应该查看最新的在线文档，以获得执行的命令和按钮的大纲。对于初学者，下面的网址中提供了有关 Apple 和 Google 的文档：Apple(http://mng.bz/z1VP)和(Google http://developer.android.com/tools/building/index.html)。

触摸输入

移动设备和桌面或 Web 的输入方式不同。移动设备通过触摸屏幕而不是鼠标和键盘完成输入。Unity 有用于处理触摸的输入功能，包括类似 Input.touchCount 和 Input.GetTouch()的代码。

可以使用这些命令编写运行在移动设备上的平台特定的代码。然而，处理输入的方式会有点困难，所以有一些代码框架可用于顺利处理触摸输入。例如，在 Unity 的 Asset Store 中搜索 Fingers 或 Lean Touch。

上面介绍了一些与构建方式无关的警告，接下来解释 iOS 和 Android 的整个构建过程。记住，这些平台偶尔会改变构建过程的一些细节。

13.3.1　设置构建工具

移动设备与开发所用的计算机不同，而这种区别使构建和部署到设备的过程更复

杂。在单击 Build 之前，需要设置不同的专用工具。

设置 iOS 构建工具

在较高的层面上，将 Unity 游戏部署到 iOS 上的过程首先需要在 Unity 中构建一个 Xcode 项目，接着使用 Xcode 将 Xcode 项目内置到 IPA(iOS app package)中。Unity 不能直接构建最终的 IPA，因为所有 iOS 应用都必须通过 Apple 的构建工具产生。这意味着必须安装 Xcode(Apple 的编程 IDE)，包括 iOS SDK。

警告 这意味着必须工作在 Mac 上——Xcode 只能运行在 OS X 上。在 Unity 中开发游戏可以在 Windows 或 Mac 上完成，但构建 iOS 应用必须在 Mac 上完成。

可以从 Apple 网站的开发部分获取 Xcode：https://developer.apple.com/xcode/。

注意 必须成为 Apple Developer Program 的会员才能在 App Store 上销售 iOS 游戏。Apple 开发程序每年的费用是 99 美元，在 https://developer.apple.com/programs/ 上登记。

一旦安装了 Xcode，就回到 Unity，将平台切换为 iOS。需要为 iOS 应用调整 Player Settings(记住，打开 Build Settings 并单击 Player Settings)。此时应该位于 Player Settings 的 iOS 选项卡上，如果需要，则单击 iPhone 选项卡。向下滚动到 Other Settings，找到 Identification。需要调整 Bundle Identifier，以便 Apple 正确识别该应用。

注意 iOS 称之为 Bundle Identifier，Android 称之为 Package Name，但在两个平台上，它们的工作方式是相同的。标识符应该遵循和其他代码包一样的约定：com.companyname.productname，且字母都小写。

另一个应用到 iOS 和 Android 的重要设置是 Version(即应用的版本号)。然而，除此之外的大多数设置是平台特定的，例如，iOS 增加了一个额外的构建版本号，独立于主版本号。还有一种 Scripting Backend 设置，之前一般使用 Mono，但新的 IL2CPP 后端可以支持平台更新，例如 64 位二进制。

现在单击 Unity 中的 Build。选择构建文件的位置，接着在这个位置生成 Xcode 项目。如果有需要，可以直接修改 Xcode 项目(一些简单的修改可以通过 postbuild 脚本完成)。

现在先打开 Xcode 项目，构建文件夹中包含很多文件，双击.xcodeproj 文件(它有一个蓝图图标)。Xcode 将打开并加载这个项目，虽然 Unity 已经考虑到项目中大多数需要的设置，但仍需要调整所使用的自动配置文件(provisioning profiles)。

iOS 自动配置文件

在 iOS 开发的所有方面中，自动配置文件是最常修改且最不寻常的文件。简言之，

这些文件用于识别和验证。Apple 严格控制哪个应用可以运行在哪个设备上，提交到 Apple 进行审批的应用使用专用的自动配置文件，该文件允许应用在 App Store 上工作，而开发中的应用使用特定于注册设备的自动配置文件。

需要将 iPhone 的 UDID(针对设备的 ID)和应用的 ID(Unity 中的 Bundle Identifier)添加到 iOS Dev Center(给 iOS 开发人员使用的 Apple 网站)的控制面板。对于这个过程的完整解释，请访问：https://developer.apple.com/support/code-signing。(见图 13-6)



图 13-6　生成自动配置文件

确保顶部的 Scheme Destination 设置为 iOS 设备，而不是模拟器(Unity 项目只在设备上工作，不在模拟器上工作，如果出错，一些构建选项就会灰显)。Xcode 将尝试自动设置签名配置文件，但如果需要，可以手动调整。在 Xcode 左边的项目列表中选择应用。这样，一些关于所选项目的选项卡将会显示出来。在 Build Settings 中向下滚动到 Code Signing，设置自动配置文件(如图 13-7 所示)。



图 13-7　Xcode 中的自动配置文件设置

一旦自动配置文件设置完毕，就开始构建应用。在 Product 菜单中，选择 Run 或 Archive。在 Product 菜单中包括许多选项，包括诱人的 Build，但对于我们的目标，有用的两个选项是 Run 或 Archive。Build 生成可执行文件，但不将这些文件绑定到 iOS，而 Run 和 Archive 完成这个操作：

- Run 在使用 USB 连接线连接到计算机的 iPhone 上测试应用。
- Archive 将创建应用包，可以将该应用包发送到其他注册的设备上(Apple 称其为“ad-hoc 发布”)。

Archive 并不是直接创建应用包，而是创建一个介于原始代码文件和 IPA 之间的中间级别的包。所创建的归档文件在 Xcode 的 Organizer 窗口中列出。在该窗口中，选择生成的归档文件，单击右边的 Export，之后系统会询问是将应用发布到商店上还是 ad hoc。

如果选择 ad hoc 发布，将得到一个可以发送给测试者的 IPA 文件。可以直接将该文件发送给测试者，并通过 iTunes 安装，但使用 TestFlight(https://developer.apple.com/testflight/)处理发布和安装测试构建将更方便。

设置安卓构建工具

与 iOS 应用不同，Unity 可以直接生成 APK(Android Application Package)。这需要将 Unity 指向 Android SDK，Android SDK 包括一个必要的编译器。从 Android 网站下载 Android SDK，接着在 Unity Preferences 中选择这个 SDK 的位置(如图 13-8 所示)。可以通过以下网站下载 SDK：

http://developer.android .com/sdk/index.html

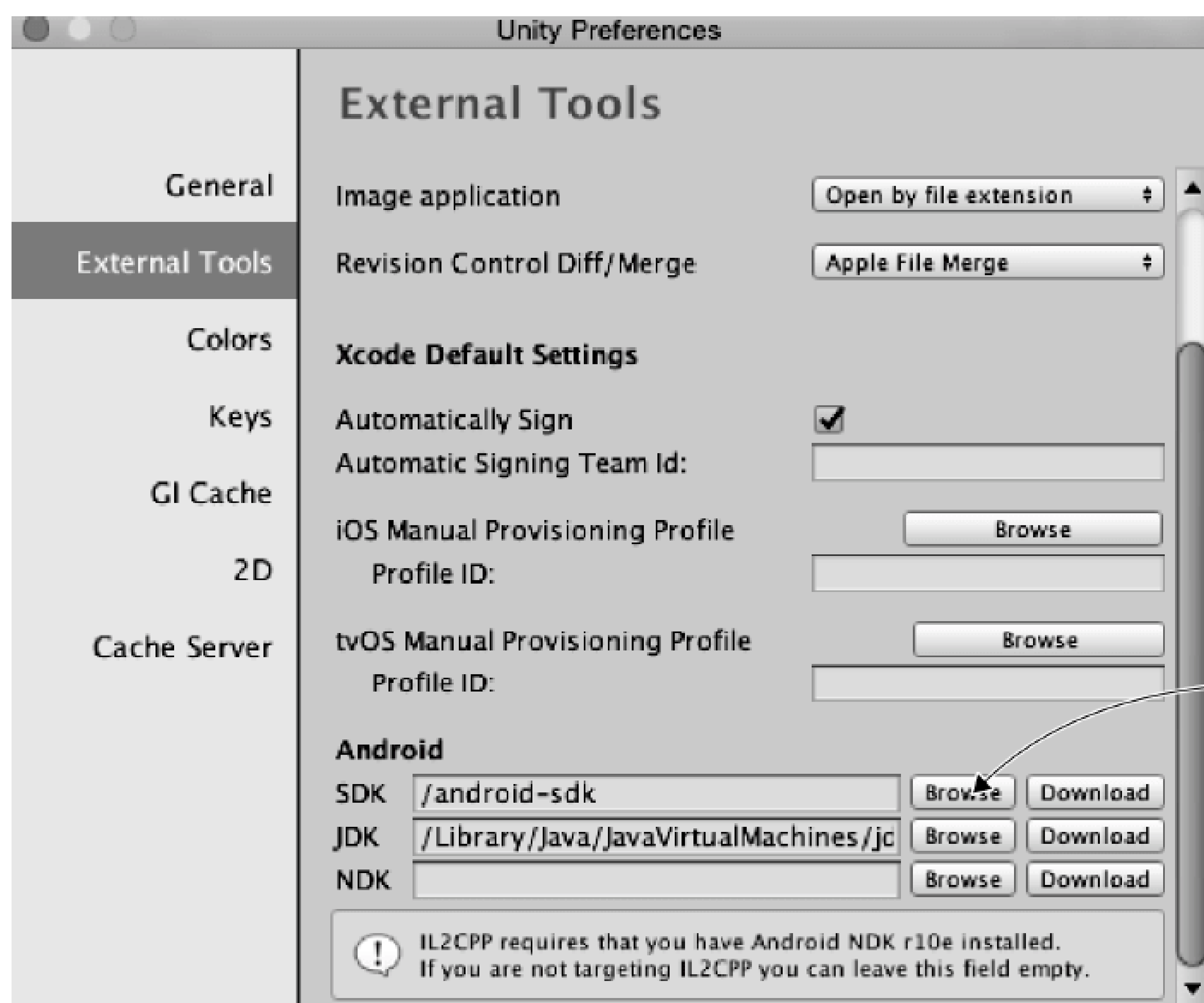


图 13-8　将 Unity Preferences 设置为指向 Android SDK

提示　虽然只需要命令行 SDK 进行基本的构建，但是可以下载 Android Studio。本章后面将使用这个工具。

在 Unity Preferences 中设置 Android SDK 之后，需要像设置 iOS 那样指定 Bundle Identifier。在 Player Settings 中找到 Bundle Identifier，将它设置为 com.companyname.productname。接着单击 Build 开始处理。和其他构建一样，它将询问在何处保存文件。接着在该位置创建 APK 文件。

有了应用包后，必须将它安装到一个设备上。可以从 Web 下载文件或通过 USB 连接线连接到计算机，将文件传输到 Android 手机上。如何将文件传输到手机的细节对于不同的设备而言是不同的，一旦上传到手机上，就可以使用文件管理器安装它。我们不了解文件管理器没有内置到 Android 中的原因，但可以从 Play Store 免费安装它。在文件管理器中导航到 APK 文件并安装它。

可以看出，Android 的基本构建过程比 iOS 更简单。但自定义构建和实现插件比 iOS 更复杂，在后面的 13.3.3 节中将介绍这些内容，下面先介绍贴图压缩。

13.3.2　贴图压缩

资源会占用大量内存，尤其是包括贴图的资源。为了减小文件的大小，可以以一些方式压缩资源，每种压缩方式都有其优缺点。因此需要调整 Unity 压缩资源的方式。

在移动设备上管理贴图压缩是有必要的，但从技术上而言，在其他平台上也经常需要压缩贴图。但出于一些不同的原因(主要原因是那些平台的技术更成熟)，不必太关注其他平台上的压缩。在移动设备上，需要更关注贴图压缩，因为设备对这些细节更敏感。

Unity 可以自动压缩贴图，而大多数开发工具需要自己压缩图像，但在 Unity 中通常导入未压缩的图像，接着 Unity 在导入设置中为图像应用图像压缩(如图 13-9 所示)。

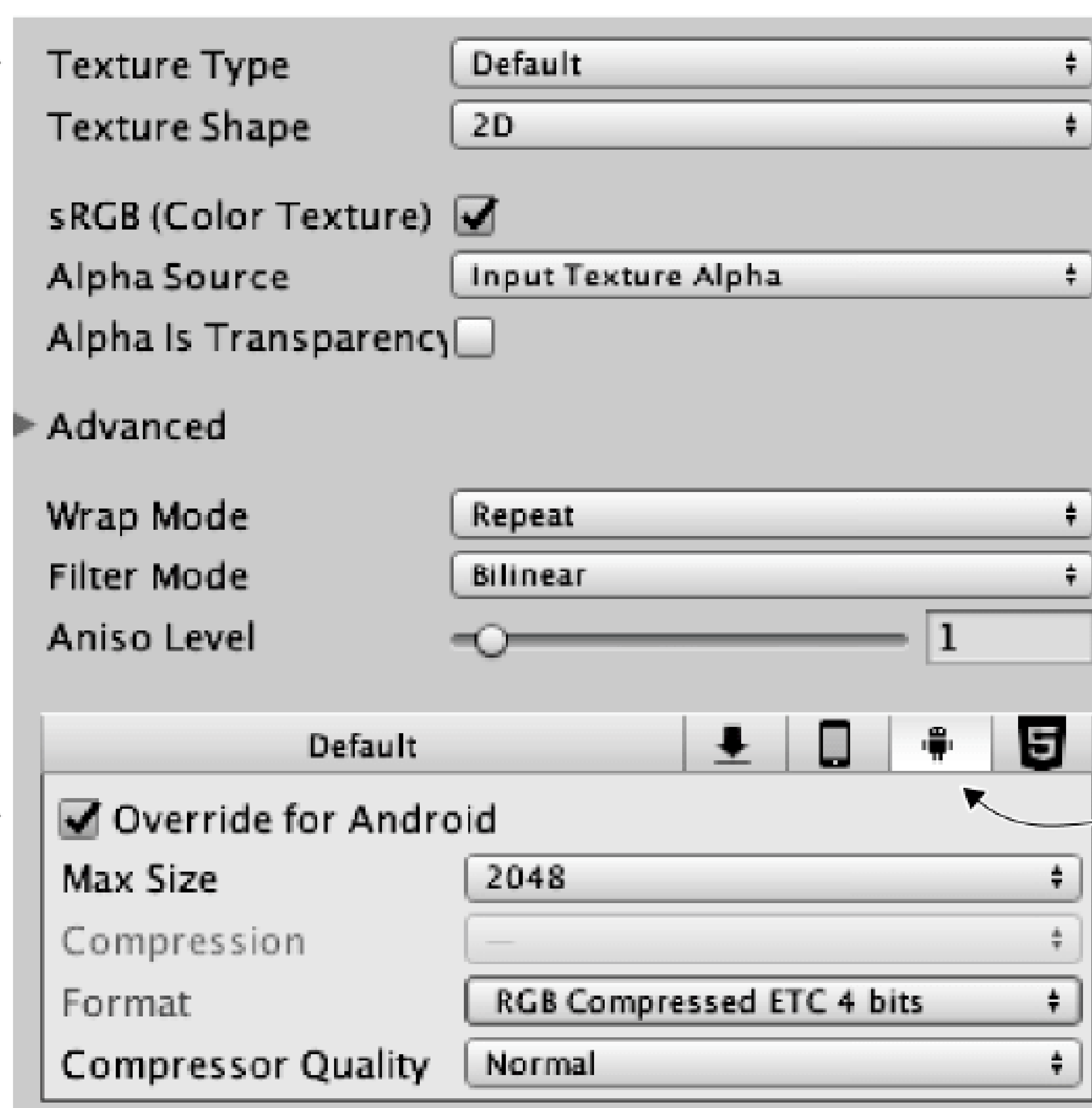


图 13-9　Inspector 中的贴图压缩设置

不同平台上的压缩设置是不同的，因此当切换平台时，Unity 会重新压缩图像。最初，这些压缩设置是默认的，可能需要在这特定的平台上为特定的图像调整它们。在 Android 上图像压缩更棘手，这主要归因于 Android 设备的存储碎片：因为所有的 iOS 设备都使用基本相同的视频硬件，iOS 应用可以从它们的图形芯片中获得贴图压缩优化。Android 应用享受不了硬件一致性的好处，因此它们的贴图压缩目标针对最低的通用标准。

具体而言，所有的 iOS 设备使用 PowerVR GPUs，因此 iOS 应用可以使用优化的 PVR 贴图压缩。一些 Android 设备也使用 PowerVR 芯片，但经常使用 Qualcomm 的 Adreno 芯片、ARM 的 Mali GPUs，或其他芯片。结果 Android 应用通常依赖爱立信贴图压缩(Ericsson Texture Compression，ETC)，这是所有 Android 设备都支持的一种压缩算法。Unity 默认给 Android 上的贴图使用 ETC2(更高级的第 2 版)。

这个默认设置适合于大多数情况，但是如果需要调整纹理上的压缩，请调整如图 13-6 所示的设置。单击 Android 图标选项卡，覆盖该平台的默认设置，然后使用 Format 菜单(不是 Compression 菜单)来选择特定的压缩格式。特别是，某些关键图像需要解压，虽然它们的文件尺寸会大得多，但是图像质量会更好。只要压缩了大部分的纹理，且选择逐个文件进行解压缩，增加的文件尺寸不会大。

讨论完贴图压缩的调整后，最后一个移动开发主题是开发本地插件。

13.3.3　开发插件

Unity 内置了很多功能，但那些功能大多限于所有平台都有的特性。而要利用平台特定工具包(例如，Android 上的 Play Game Services)，通常需要安装 Unity 的附加插件。

提示　有各种用于 iOS 和 Android 特性的预制移动插件，附录 D 列出了一些可以获取移动插件的地方。这些插件的操作方式如本节所述，但插件代码已编写好。

和移动插件的通信过程与和浏览器的通信过程类似。在 Unity 端，使用特定命令调用插件内的方法。在插件端，插件可以使用 SendMessage()将消息发送给 Unity 场景中的对象。具体的代码在不同平台上是不同的，但基本理念通常一样。

警告　正如初始构建过程，开发移动插件的过程也频繁变化——不是该过程的 Unity 端，而是本地代码部分。下面将概述它，但读者应该查阅最新的在线文档。

用于两种平台的插件在 Unity 中放在相同的位置。如果需要，在 Project 视图中创建一个名为 Plugins 的文件夹，然后，在 Plugins 文件夹内部为 Android 和 iOS 分别创建一个文件夹。一旦放入 Unity 中，插件文件也指定了它们所应用的平台。通常情况下，Unity 会自动指定这个设置(iOS 插件设置为 iOS，Android 插件设置为 Android 等)，但是如果有必要，在 Inspector 中查找这些设置。

iOS 插件

“插件”通常是 Unity 调用的一些本地代码。首先在 Unity 中创建一个脚本，用于处理本地代码，将该脚本文件命名为 TestPlugin(如代码清单 13.5 所示)。

代码清单 13.5 从 Unity 中调用 iOS 本地代码的 TestPlugin 脚本

```
using UnityEngine;
using System;
using System.Collections;
using System.Runtime.InteropServices;

public class TestPlugin : MonoBehaviour {
  private static TestPlugin _instance;

  public static void Initialize() {
    if (_instance != null) {
        Debug.Log("TestPlugin instance was found. Already initialized");
        return;
    }
    Debug.Log("TestPlugin instance not found. Initializing...");

    GameObject owner = new GameObject("TestPlugin_instance");
    _instance = owner.AddComponent<TestPlugin>();
    DontDestroyOnLoad(_instance);
  }

  #region iOS
  [DllImport("__Internal")]
  private static extern float _TestNumber();

  [DllImport("__Internal")]
  private static extern string _TestString(string test);
  #endregion iOS

  public static float TestNumber() {
     float val = 0f;
     if (Application.platform == RuntimePlatform.IPhonePlayer)
        val = _TestNumber();
     return val;
  }

  public static string TestString(string test) {
     string val = "";
     if (Application.platform == RuntimePlatform.IPhonePlayer)
        val = _TestString(test);
     return val;
  }
}
```

注释：
- 在这个静态函数中创建对象，因此不必在编辑器中创建它（指向 Initialize()）
- 标识代码部分的标签，标签本身不做任何事情（指向 #region iOS）
- 引用 iOS 代码中的函数（指向 DllImport 声明）
- 如果平台是 IPhonePlayer，就调用这个函数（指向 val = _TestNumber();）

首先，注意静态函数 Initialize()创建了场景中的永久对象，因此不必在编辑器中手动创建该对象。之前还未介绍过从头创建对象的代码，因为大多数情况使用预设更简单，但本例中在代码中创建对象更为简洁(因此可以使用插件脚本，不需要编辑场景)。

此处主要的疑惑在于 DLLImport 和 static extern 命令。这些命令告诉 Unity 连接所提供的本地代码中的函数。接着可以在这个脚本的方法中使用那些引用的函数(进行检查，来确认代码运行在 iPhone/iOS 上)。

接下来使用这些插件函数并测试它们。创建一个称为 MobileTestObject 的脚本，在场景中创建一个空对象，并将该脚本(代码清单 13.6 所示)附加到这个空对象上。

代码清单 13.6 在 MobileTestObject 中使用插件

```
using UnityEngine;
using System.Collections;

public class MobileTestObject : MonoBehaviour {
  private string _message;

  void Awake() {
    TestPlugin.Initialize();          // 开始时初始化插件
  }

  // Use this for initialization
  void Start() {
    _message = "START: " + TestPlugin.TestString("ThIs Is A tEsT");
  }

  // Update is called once per frame
  void Update() {

    // Make sure the user touched the screen
    if (Input.touchCount==0){return;}

    Touch touch = Input.GetTouch(0);          // 响应触摸输入
    if (touch.phase == TouchPhase.Began) {
      _message = "TOUCH: " + TestPlugin.TestNumber();
    }
  }

  void OnGUI() {
    GUI.Label(new Rect(10, 10, 200, 20), _message);          // 在屏幕角落显示消息
  }
}
```

代码清单中的脚本初始化了插件对象，并调用插件方法以响应触摸输入。一旦在设备上运行，不管何时单击屏幕，屏幕角落的测试消息都会发生变化。

最后还需要做的事情是编写 TestPlugin 引用的本地代码。iOS 设备上的代码使用 Objective C 和/或 C 编写，因此需要 a.h 头文件和 a.mm 实现文件。如前所述，它们需要放在 Project 视图的 Plugins/iOS/文件夹中。在该文件夹中创建 TestPlugin.h 和 TestPlugin.mm，在.h 文件中编写代码清单 13.7 中的代码。

代码清单 13.7　用于 iOS 代码的 TestPlugin.h 头文件

```
#import <Foundation/Foundation.h>

@interface TestObject : NSObject {
NSString* status;
}

@end
```

查找关于 iOS 编程的解释，以了解 TestPlugin.h 头文件的作用，有关 iOS 编程的解释超出本书的讨论范围。在.mm 文件中编写代码清单 13.8 中的代码。

代码清单 13.8　TestPlugin.mm 实现文件

```
#import "TestPlugin.h"

@implementation TestObject
@end

NSString* CreateNSString (const char* string)
{
if (string)
return [NSString stringWithUTF8String: string];
else
return [NSString stringWithUTF8String: ""];
}

char* MakeStringCopy (const char* string)
{
if (string == NULL)
return NULL;

char* res = (char*)malloc(strlen(string) + 1);
strcpy(res, string);
return res;
}

extern "C" {
    const char* _TestString(const char* string) {
        NSString* oldString = CreateNSString(string);
        NSString* newString = [oldString uppercaseString];
        return MakeStringCopy([newString UTF8String]);
    }

    float _TestNumber() {
        return (arc4random() % 100)/100.0f;
    }
}
```

同样，这段代码的解释也超出了本书的讨论范围。注意，代码中的很多字符串函数在 Unity 描述的字符串数据和本地代码使用的字符串之间转换。

提示　这个示例只是从 Unity 到插件的单向通信。本地代码也可以使用 UnitySendMessage() 方法向 Unity 通信。可以将消息发送给场景中指定名称的对象，在初始化时插件会创建 TestPlugin_instance，用于发送消息。

本地代码准备就绪后，就可以构建 iOS 应用，在设备上测试了。轻触屏幕，观察显示的数字。以上介绍的是创建 iOS 插件的方式，接下来将介绍 Android 插件。

Android 插件

为了创建 Android 插件，Unity 这边的处理大致一样。不必修改 MobileTestObject，在 TestPlugin 中添加代码清单 13.9 中的内容。

代码清单 13.9　修改 TestPlugin 以使用 Android 插件

```
...
   #region iOS
   [DllImport("__Internal")]
   private static extern float _TestNumber();

   [DllImport("__Internal")]
   private static extern string _TestString(string test);
   #endregion iOS

#if UNITY_ANDROID
   private static Exception _pluginError;
   private static AndroidJavaClass _pluginClass;
   private static AndroidJavaClass GetPluginClass() {
      if (_pluginClass == null && _pluginError == null) {
         AndroidJNI.AttachCurrentThread();
        try {
         __pluginClass = new AndroidJavaClass("com.testcompany.testplugin.
      TestPlugin");
        } catch (Exception e) {
          _pluginError = e;
        }
      }
      return _pluginClass;
   }

   private static AndroidJavaObject _unityActivity;
   private static AndroidJavaObject GetUnityActivity() {
      if (_unityActivity == null) {
        AndroidJavaClass unityPlayer = new AndroidJavaClass("com.unity3d.
    player.UnityPlayer");
        _unityActivity = unityPlayer.
   GetStatic<AndroidJavaObject>("currentActivity");
    }
    return _unityActivity;
   }
#endif

   public static float TestNumber() {
      float val = 0f;
```

（注释：Unity 提供的 AndroidJNI 功能；我们编写的类名，可以根据需要修改这个名称；Unity 为 Android 应用创建活动）

```
      if (Application.platform == RuntimePlatform.IPhonePlayer)
        val = _TestNumber();
#if UNITY_ANDROID
      if (!Application.isEditor && _pluginError == null)
        val = GetPluginClass().CallStatic<int>("getNumber");
#endif
      return val;
    }

    public static string TestString(string test) {
      string val = "";
      if (Application.platform == RuntimePlatform.IPhonePlayer)
        val = _TestString(test);
#if UNITY_ANDROID
      if (!Application.isEditor && _pluginError == null)
        val = GetPluginClass().CallStatic<string>("getString", test);
#endif
      return val;
    }
}
```

调用 plugin.jar 中的函数

注意，大多数添加的代码都在平台定义的 UNITY_ANDROID 内部。如前面章节所述，这些编译器指令使代码只应用于特定的平台，在其他平台被忽略。iOS 代码不会执行打断其他平台的操作(不做任何事情，也不会产生错误)，而当 Unity 设置为 Android 平台时，才会编译 Android 插件的代码。

特别要注意对 AndroidJNI 的调用。它是 Unity 连接到本地 Android 的系统。另一个可能让人疑惑的单词是 activity(活动)，在 Android 应用中，活动就是一个应用进程。Unity 游戏是 Android 应用的一个活动，因此当插件代码访问该活动时，需要传入该活动。

最后，需要本地的 Android 代码。iOS 代码用 Objective C 和 C 等语言编写，而 Android 用 Java 编写。但不能简单地为插件提供原始 Java 代码，插件必须是打包 Java 代码的 jar。另外，Android 编程的细节已超出了本书介绍 Unity 的范围，这里只简单介绍基础知识。首先，如果下载 Android SDK 时没有安装 Android Studio，现在就安装它。

图 13-10 说明了在 Android Studio 中建立插件项目的步骤。首先，选择 Start a New Project。在出现的配置窗口中，将其命名为 TestPluginProj。对于这个测试，公司域是什么并不重要，但是要注意项目的位置，因为需要找到它。单击 Next 继续，目前，Minimum SDK 并不重要，但是选择 Add No Activity(因为这是一个插件，而不是独立的 Android 应用)。

一旦单击 Finish，稍等一会就会建立新项目。一旦编辑器视图出现，就选择 File | New | New Module，添加一个库。在配置窗口中选择 Java Library，将其命名为 test-plugin，然后单击 Edit，将 Java 包名称更改为 com.testcompany.testplugin。最后指定类名 TestPlugin。现在打开 Project 视图(它是左侧边缘的一个按钮)，展开 test-plugin，然后双击 TestPlugin 类来打开它。

TestPlugin 目前是空的，因此在其中编写插件函数。代码清单 13.10 显示了插件的 Java 代码。

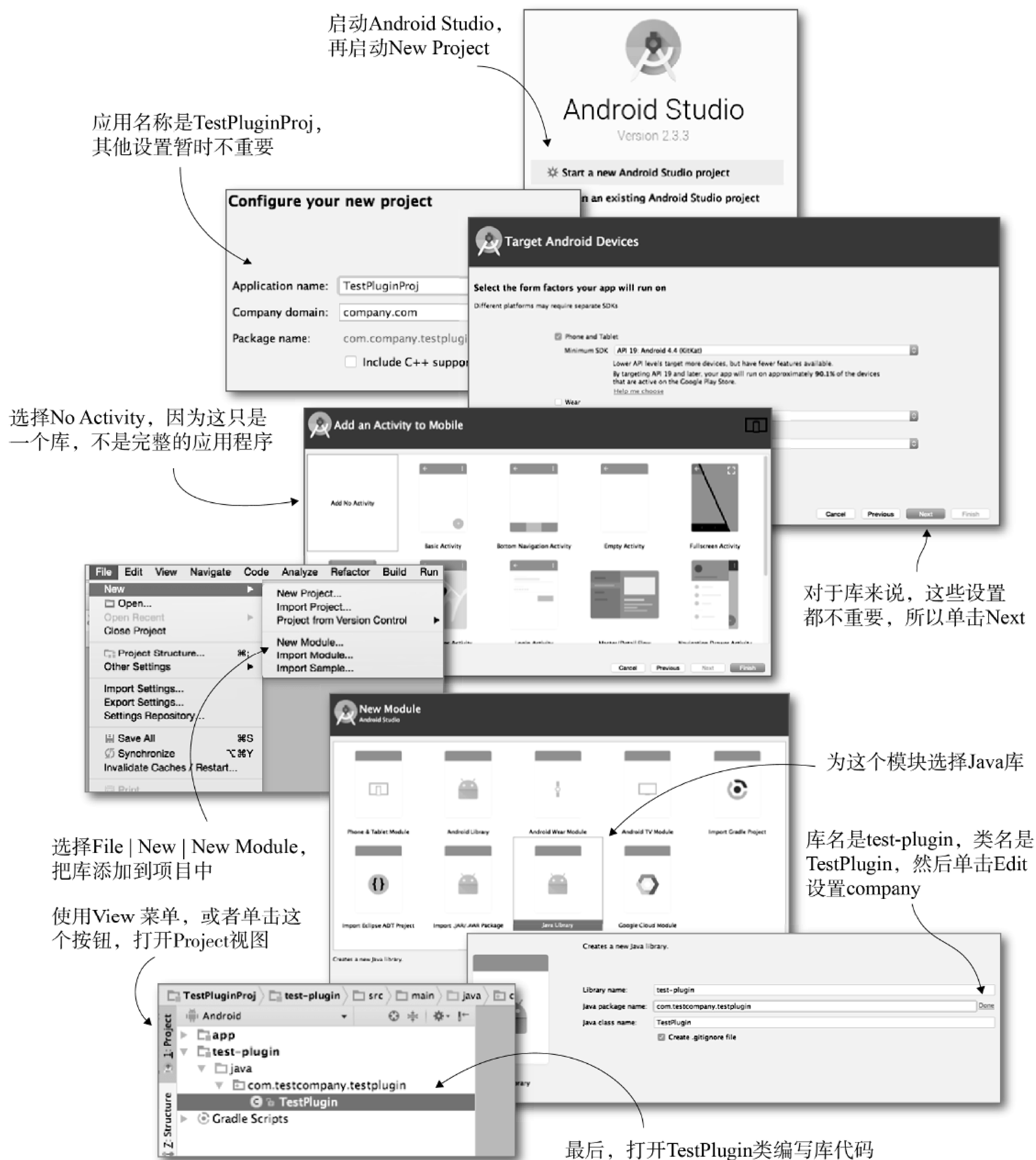


图 13-10　安装 Android Studio，建立插件

代码清单 13.10　编译为 JAR 的 TestPlugin.java

```
package com.testcompany.testplugin;

public class TestPlugin {
    private static int number = 0;

    public static int getNumber() {
        number++;
        return number;
    }
```

```
        public static String getString(String message) {
            return message.toLowerCase();
        }
    }
```

现在可以将代码打包到 JAR 中。在顶部菜单中，选择 Build > Build APK。尽管有这个名称，但这也会构建库，可以忽略 apk 本身。构建完成后，转到计算机上的项目，找到/test-plugin/build/libs/下的 test-plugin.jar。将 JAR 拖到 Unity 的 Android 插件文件夹中，以导入它。

Android 的清单(manifest)和资源文件夹

这个简单的测试插件还不需要使用清单和资源文件夹，但 Android 插件通常必须编辑清单文件。所有 Android 应用都由主配置文件 Android-Manifest.xml 控制，如果不提供这个文件，Unity 就会创建一个基本的清单文件，也可以在 Plugins/Android/中的插件 JAR 旁边放置一个清单，来手动提供它。

当构建 Android 应用时，Unity 将生成的清单文件放置在 StagingArea/AndroidManifest.xml 的 Temp 文件夹内。复制该清单文件，手动编辑它(下载的代码中也包括了一个示例清单文件)。

类似地，还有一个 res 文件夹，在其中可以放置自定义图标等资源，可以在 Android 插件文件夹中创建 res 文件夹。

很多开发 Android JAR 的细节在 JAR 文件位于 Plugins/Android 中后，构建游戏，在设备上安装它。只要轻触屏幕，消息都会改变。和 iOS 插件一样，Android 插件也可以使用 UnityPlayer.UnitySendMessage()和场景中的对象通信(Java 代码需要导入 Unity 的 Android Player 库/JAR)。

这里没有介绍，这是因为该过程太复杂而且会经常变化。如果高级开发者要为 Android 游戏开发插件，就需要查看 Android 开发者网站的文档。

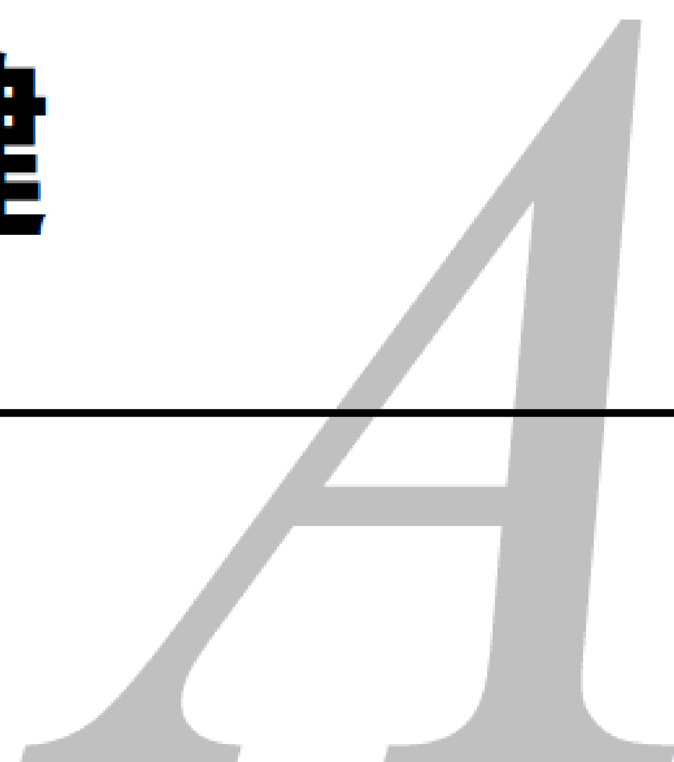
这就完成了本书的学习

祝贺你，现在你已经了解了为移动设备开发 Unity 游戏的步骤。所有平台的基本构建都很简单(只是一个按钮就可以完成)，但在不同平台上自定义应用则比较复杂。现在读者可以合上本书，开始构建自己的游戏！

13.4 小结

- Unity 可以为各种平台构建可执行的应用，包括台式机、移动设备和网站。
- 有很多可以应用到构建的设置，包括应用图标和名称这样的细节。
- Web 游戏可以和嵌入的网页交互，包括所有类型的 Web 应用。
- Unity 支持自定义插件以扩展自己的功能。

附录 A 场景导航和快捷键



Unity 通过鼠标和键盘操作，但新手并不能明显地知道如何在 Unity 中使用鼠标和键盘。特别地，最基本的鼠标和键盘输入是用于在场景中导航和查看 3D 对象。Unity 也有一些键盘命令用于常用操作。

本附录解释输入控件，有一些网页可供参考(这些网页是 Unity 在线指南的相关页面)：http://docs.unity3d.com/Documentation/Manual/SceneViewNavigation.html 和 http://docs.unity3d.com/Documentation/Manual/UnityHotkeys.html

A.1 使用鼠标进行场景导航

场景导航有三个主要的导航菜单：Move、Orbit 和 Zoom。这三种不同的运动包括按住一些 Alt(或 Mac 上的 Option)和 Control 组合键时的单击和拖动。对于一个、两个、三个按键的鼠标，具体的控件也不同，表 A-1 列出了所有控件。

表 A-1 不同鼠标的场景导航控件

导航行为	三个按键的鼠标	两个按键的鼠标	一个按键的鼠标
Move	中键单击	Alt+Command＋左键/拖动	Alt+Command+单击/拖动
Orbit	按住 Alt+左键/拖动	Alt+左键/拖动	Alt+单击/拖动
Zoom	按住 Alt+右键/拖动	Alt+右键/拖动	Alt+Ctrl+单击/拖动

注意 尽管 Unity 可以使用一个或两个按键的鼠标，但强烈建议使用三个按键的鼠标(三个按键的鼠标也适用于 Mac OS X)。

除了使用鼠标完成的一些导航操作，还有一些基于键盘的视图控件。如果按下鼠

标的右键，键盘的 WASD 按键用于以大多数第一人称游戏中通用的方式移动。在按住其他任何键时按下 Shift，可以移动得更快。但最重要的是，如果在选中对象时按下 F 键，视野将平移并缩放到该对象。如果在场景导航中迷失，通用的“安全措施”是在 Hierarchy 中选择列出的对象并按 F 键。

A.2 有用的快捷键

Unity 有一些键盘命令用于快速访问一些重要功能。最重要的快捷键是 W、E、R 和 T：这些按钮激活了变换工具 Translate、Rotate 和 Scale(如果忘了变换工具的作用，请参阅第 1 章)以及 2D Rect 工具。因为那些按键相互挨得很近，所以通常让左手停留在那些按键上，而右手操作鼠标。

除了变换工具外，还有一些快捷键。表 A-2 列出了很多在 Unity 中有用的快捷键。

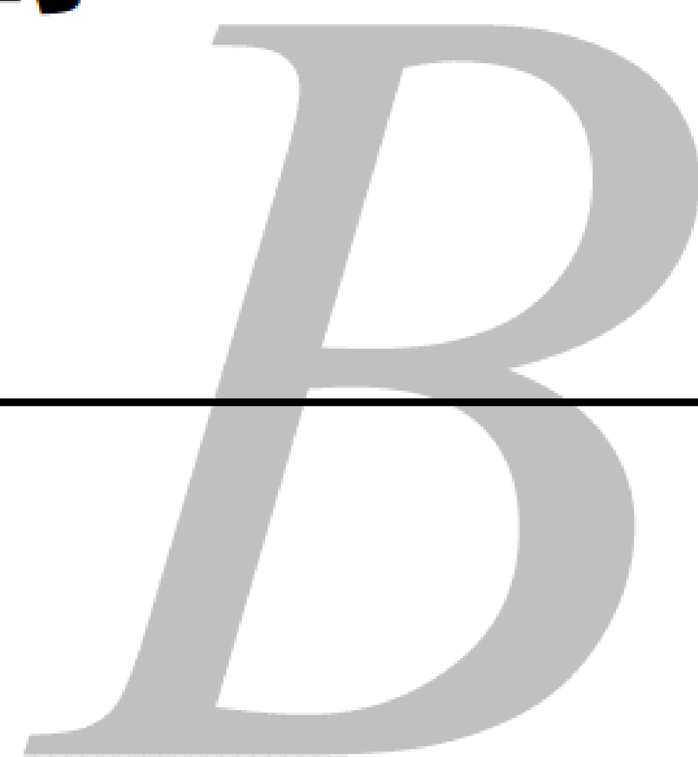
表 A-2 有用的快捷键

按 键	功 能
W	平移(移动选中的对象)
E	旋转(旋转选中的对象)
R	缩放(改变选中对象的大小)
T	矩形工具(操作 2D 对象)
F	将视野聚焦在选中的对象上
V	对齐到顶点
Ctrl/Command+Shift+N	新建 GameObject
Ctrl/Command+P	运行游戏
Ctrl/Command+R	刷新对象
Ctrl/Command+1	将当前窗口设置为 Scene 视图
Ctrl/Command+2	设置为 Game 视图
Ctrl/Command+3	设置为 Inspector 视图
Ctrl/Command+4	设置为 Hierarchy 视图
Ctrl/Command+5	设置为 Project 视图
Ctrl/Command+6	设置为 Animation 视图

Unity 也响应其他快捷键，但这些快捷键和表 A-2 列出的相比越来越不受关注。

附录 B

与 Unity 一同使用的外部工具



使用 Unity 开发游戏依赖于各种外部工具以完成不同的任务。第 1 章讨论了一个外部工具：MonoDevelop，该工具在技术上是一个独立的应用，尽管它随着 Unity 一起打包。同样，开发者依赖一系列外部工具完成 Unity 外部的工作。

这并不是说 Unity 缺少了该有的功能，而是游戏开发过程是复杂且多方面的，设计良好的软件有清晰的目标。关注点分离也做得很好，但这样必然只擅长处理开发过程中有限的方面。例如，Unity 就是擅长于将游戏的所有内容整合到一起并使之运转起来的引擎。这些内容的创建通过其他工具完成，下面介绍一些可能有用的软件类型。

B.1 编程工具

前面介绍过 MonoDevelop，它是和 Unity 一起使用的最重要的编程工具。还有一些其他编程工具可以使用，如本附录所述。

B.1.1 Visual Studio

如第 1 章所述，Unity 自带 MonoDevelop，可以在 Windows 和 Mac 上使用那个 IDE。在 Windows 上，也可以选择使用 Visual Studio。Microsoft 收购了 SyntaxTree(一家提升 Visual Studio 集成度的公司)：http://unityvs.com。

B.1.2 Xcode

Xcode 是由 Apple 提供的编程环境(特别是 IDE，也包含用于 Apple 平台的 SDK)。尽管依然在 Unity 中完成大部分工作，但需要使用 Xcode 将游戏部署到 iOS 上。该工

作通常包含使用 Xcode 中的工具调试应用或对应用进行性能分析：https://developer.apple.com/xcode/。

B.1.3 Android SDK

需要安装 Xcode，才能将应用部署到 iOS 上，同样，也需要下载 Android SDK，才能部署到 Android 上。与构建 iOS 游戏不同的是，不需要启动任何 Unity 之外的开发工具——只需要在 Unity 中设置 preferences，以指向 Android SDK：http://developer.android.com/sdk/index.html。

B.1.4 SVN、Git 或 Mercurial

任何规模适中的软件开发项目都会包含代码文件的很多复杂的修订版本，因此程序员开发了一类称为 VCS(Version Control System，版本控制系统)的软件来处理这个问题。三个最有名的系统是 Subversion(通常称为 SVN)(http://subversion.apache.org/)、Git(http://git-scm.com/)和 Mercurial(https://www.mercurial-scm.org)。如果还没有使用 VCS，强烈建议开始使用其中的一个。Unity 的项目文件夹中包含了临时文件和工作空间设置，但版本控制只限于两个文件夹：Assets(确保版本控制选择由 Unity 生成的元文件)和 Project Settings。

B.2 3D 美术应用

尽管 Unity 能处理 2D 图形(第 5 章和第 6 章都关注 2D 图形)，但它的初衷是作为 3D 游戏引擎，且拥有强大的 3D 图形特性。很多 3D 美术师都至少使用本附录中介绍的一个软件包。

B.2.1 Maya

Maya(www.autodesk.com/products/autodesk-maya/overview)是扎根于动画制作的 3D 美术和动画包。Maya 的特性集涵盖了 3D 美术师需要完成的几乎所有任务，从制作出色的影视动画到制作高效的游戏模型。可以将 Maya 中完成的 3D 动画(例如，角色行走)导出到 Unity 中。

B.2.2 3ds Max

3ds Max(www.autodesk.com/products/autodesk-3ds-max/overview)是另一个广泛使用的 3D 美术和动画包，它提供了一个可以和 Maya 相媲美的特性集和工作流。3ds Max

只能运行在 Windows 上(而其他工具，包括 Maya，是跨平台的)，但它通常用于游戏行业。

B.2.3　Blender

Blender(www.blender.org/)尽管不像 3ds Max 或 Maya 那样广泛用于游戏行业，但它也可以与这两个应用相媲美。Blender 也覆盖了大多数 3D 美术任务，而其最突出的是，Blender 是开源的。所有平台都可以免费运行它。

B.2.4　SketchUp

这是一个非常易用的建模工具，特别适合于建筑和建筑元素。不像以前的工具，SketchUp (www.sketchup.com)没有覆盖大部分的 3D 美术任务；相反，它侧重于简化建筑物和其他简单形状的建模。这个工具在游戏开发的白盒和关卡编辑中很有用。

B.3　2D 图像编辑器

2D 图像对所有游戏都很重要，因为它们在 2D 游戏中直接显示，或显示为 3D 模型表面的贴图。游戏开发中有一些常用的 2D 图形工具，如本附录所述。

B.3.1　Photoshop

Photoshop(www.photoshop.com)无疑是应用最广泛的 2D 图像应用。Photoshop 中的工具能用于润色已有的图像、应用图像滤镜，甚至从头绘制图像。Photoshop 支持数十种不同的文件格式，包括 Unity 使用的所有图像格式。

B.3.2　GIMP

GIMP(www.gimp.org)是 GNU Image Manipulation Program 的首字母缩写，是一个广为人知的开源 2D 图形应用。GIMP 的特性和可用性与 Photoshop 一样，但它依然是一个有用的图像编辑器，且不需要付费！

B.3.3　TexturePacker

前面提及的工具都可以用于游戏开发之外的领域，但 TexturePacker 则仅对游戏开发有用。该工具非常擅长装配 2D 游戏中使用的精灵表。如果开发 2D 游戏，就可以尝试 TexturePacker(www.codeandweb.com/texturepacker)。

B.3.4 Aseprite, Pyxel Edit

Pixel art 是最容易识别的 2D 游戏美术风格之一，是很好的像素化美术工具。Photoshop 在技术上也可以用于像素化美术，但它并不专注于这一任务。此外，动画功能在 Aseprite (www.aseprite.org/)和 Pyxel Edit (www.pyxeledit.com)中更突出。

B.4 音频软件

有很多令人眼花缭乱的音频生产工具，包括声音编辑器(处理原始波形)和音序器(使用音符序列合成音乐)。对于可用的音频软件，本节关注两个主要的声音编辑工具(列表未包括的其他例子有 Logic、Abletom 和 Reason)。

B.4.1 Pro Tools

这个音频软件(www.avid.com/US/products/family/Pro-Tools)有很多有用的特性，且被无数的音乐制作人和音频工程师认为是业界标准。它经常用于各种类型的专业音频工作，包括游戏开发。

B.4.2 Audacity

尽管 Audacity(http://audacity.sourceforge.net/)并不用于专业的音频工作，但它是一个用于小规模音频工作的方便的音频编辑器，例如，准备短小的声音文件作为游戏中的音效。它非常适合于寻找开源声音编辑软件的开发人员。

附录 C

在 Blender 中建模一个板凳

C

第 2 章和第 4 章创建了一个关卡，其中包含了一些大的平面墙和地板。但没有介绍更多的对象，例如，房间中有趣的家具。它们可以在外部 3D 美术应用中构建 3D 模型。回想第 4 章介绍的定义：3D 模型是游戏中的网格对象(也就是三维图形)。本附录将展示如何创建一个简单的板凳网格对象(如图 C-1 所示)。

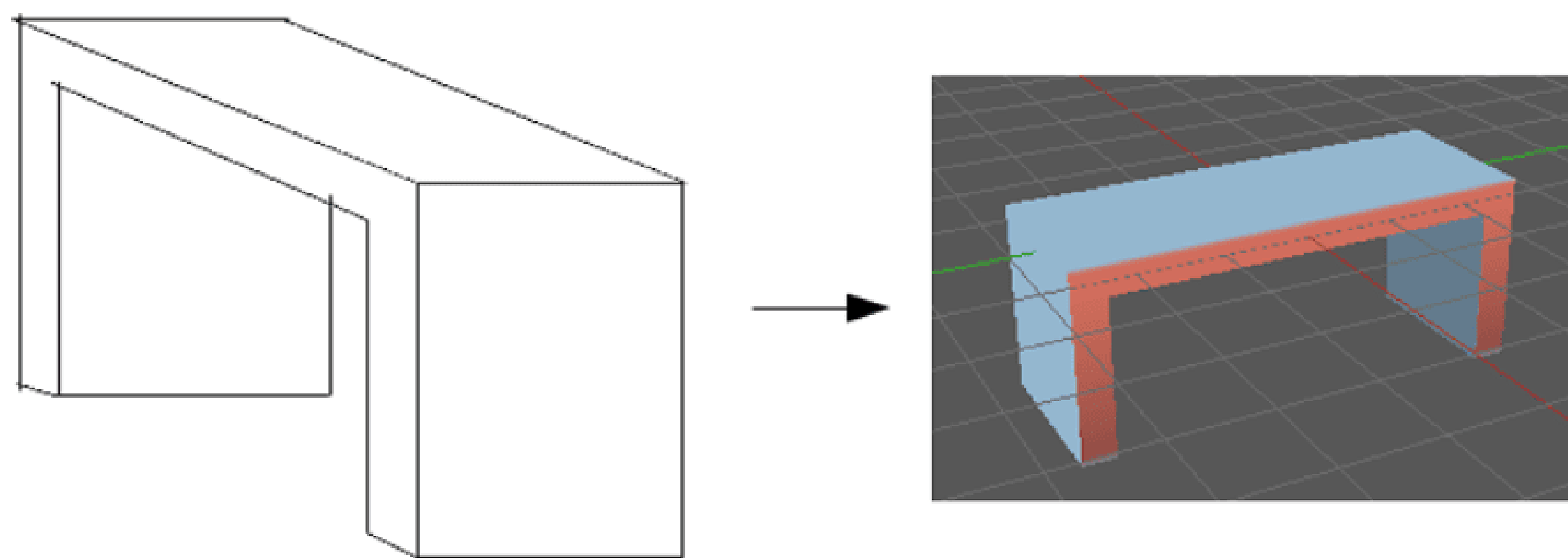


图 C-1　将建模的简单板凳

虽然附录 B 列出了一些 3D 美术工具，但本练习将使用 Blender，因为它是开源的，所有读者都可以访问它。接下来将在 Blender 中创建网格对象，并导出为可以在 Unity 中工作的美术资源。

提示　建模是一个大主题，但这里仅讨论将覆盖创建板凳的基本建模功能。如果在本附录结束后，还想进一步学习建模的相关知识，可以查看有关这个主题的一些书籍和教程(首先应阅读 www.blender.org 上的学习资源)。

警告　本附录使用的是 Blender 2.67，因此解释和屏幕截图都来自这个版本的软件。更新版本的 Blender 经常发布，一些按钮的位置或命令名可能会略有修改。

C.1　构建网格几何体

启动 Blender，初始的默认屏幕如图 C-2 所示，在屏幕中间有一个立方体。使用鼠标中键管理摄像机视角：单击并拖动用于翻转，Shift+单击拖动用于平移，而 Control+单击拖动用于缩放。

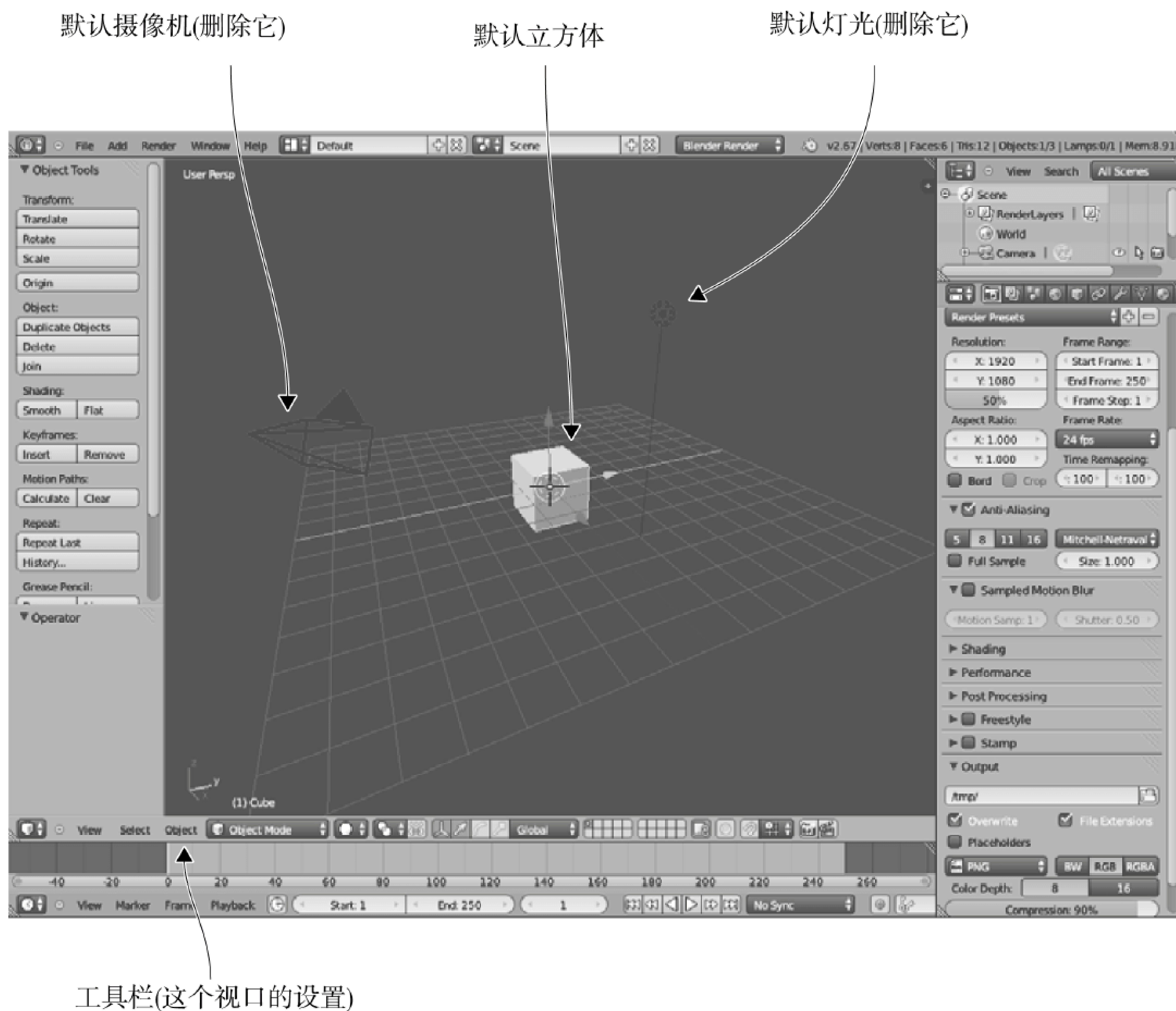


图 C-2　Blender 的初始默认屏幕

Blender 开始时使用 Object 模式，顾名思义，这种模式用于管理所有对象，在场景中移动它们。为了编辑单个对象的细节，必须切换到 Edit 模式。图 C-3 展示了使用的菜单(只有选中某个对象，菜单中才会出现 EditMode，而 Blender 以选择的对象开始)。同样，第一次切换到 Edit 模式时，Blender 会设置为 Vertex Selection 模式，可以通过按钮在 Vertex、Edge 和 Face Selection 模式之间切换(参见图 C-4)。不同的选择模式允许选择不同的网格元素。

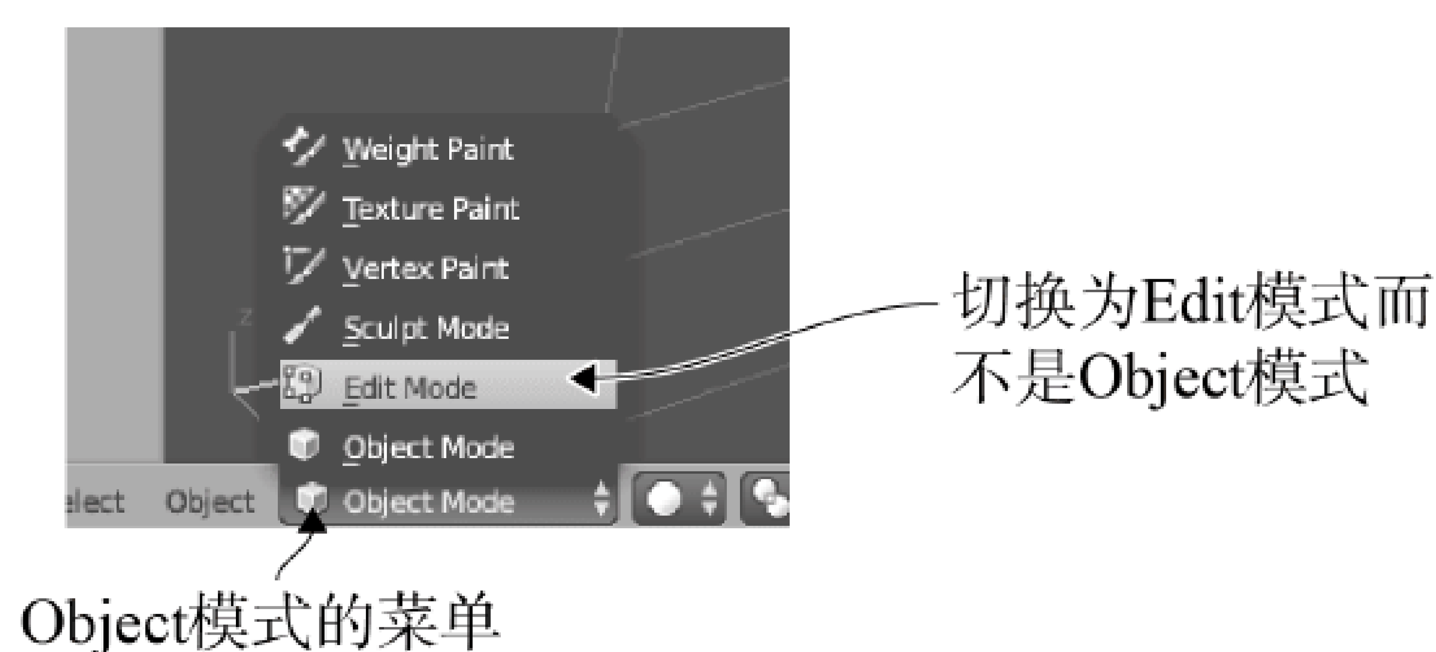


图 C-3 将 Object 模式切换为 Edit 模式的菜单

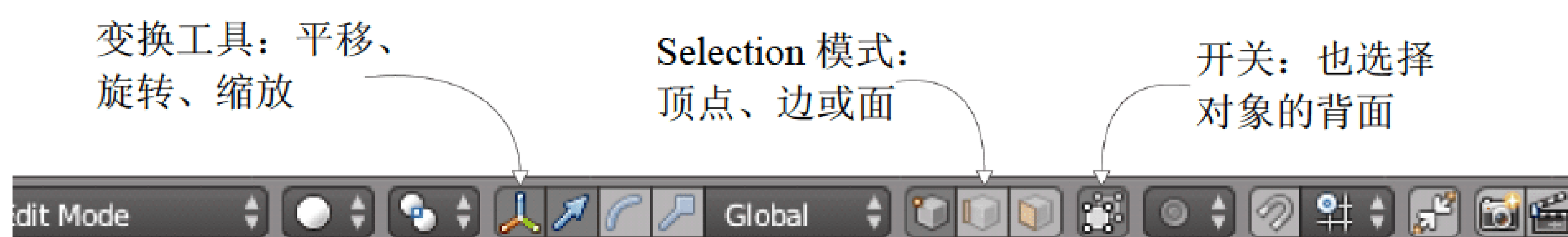


图 C-4 视口底部的控件

定义 网格元素是组成网格几何体的顶点、边和面——换句话说，就是各个角点、连接点的线、在连接线之间填充的形状。

Blender 中基础的鼠标和键盘快捷键

图 C-4 中描绘的是变换工具。与 Unity 一样，变换是平移、旋转和缩放。第一个按钮切换 Transform Gizmo(场景中的箭头)开和关，建议一直打开 Gizmo，否则只能通过键盘快捷键访问变换工具。Blender 中的键盘快捷键通常出人预料，鼠标的功能也是这样。

例如，虽然鼠标中键用于操作摄像机很直观，但是在场景中选择元素是通过鼠标右键完成的(在大多数应用中，使用鼠标左键选择对象)。更古怪的是，对盒子的选择通过按下 B 键，接着单击左键和拖动完成。当单击元素时通过按下 Shift 键添加选择(而不是替换选择)，而通过按下 A 键清除选择。

这些是使用 Blender 的基本控件，现在介绍一些用于 Edit 模式的功能。首先，将立方体缩放成一个长且薄的模板。选择模型的每个顶点(确保也选择了对象另一边的顶点)，接着切换为 Scale 工具。单击-拖动 Z 轴的蓝色箭头，以在垂直方向缩小，接着单击-拖动 Y 轴的绿色箭头，向一侧缩放(如图 C-5 所示)。

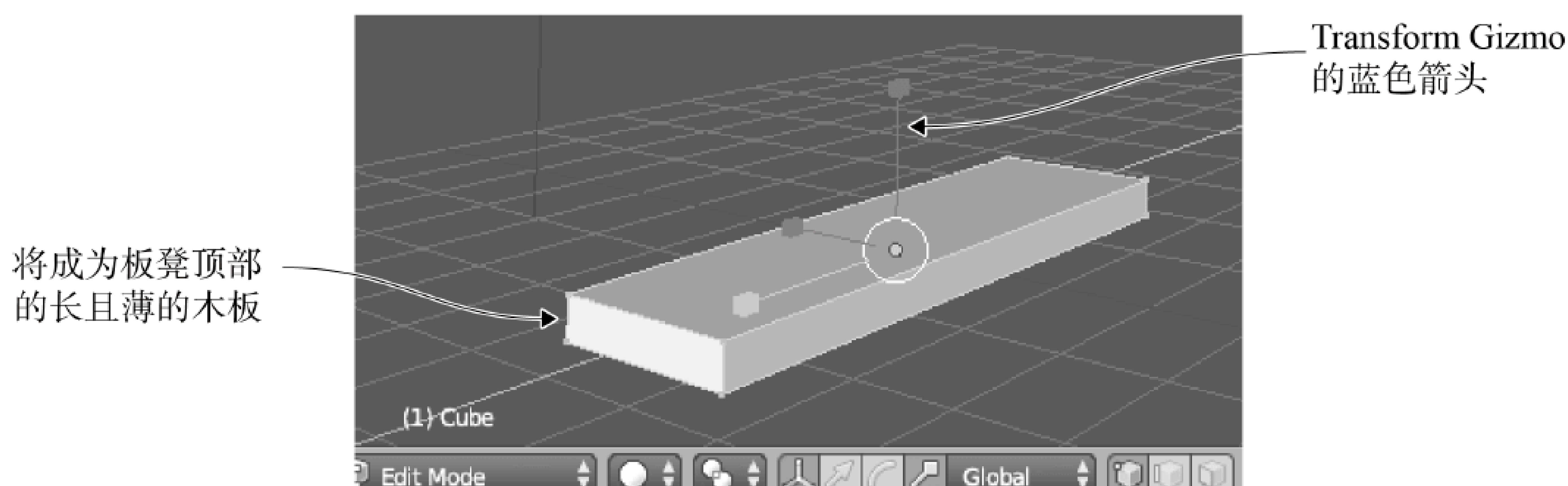


图 C-5 将网格缩放为一个长且薄的木板

切换为 Face Selection 模式(使用图 C-4 中的按钮)，选择木板的两个小的末端。现在单击视图底部的 Mesh 菜单，并选择 Extrude Individual(如图 C-6 所示)。随着鼠标移动，将看到木板末端增加了一些额外部分，稍微移开它们并左击确认。额外的部分仅有板凳脚那么宽，这就提供了便于工作的额外几何体。

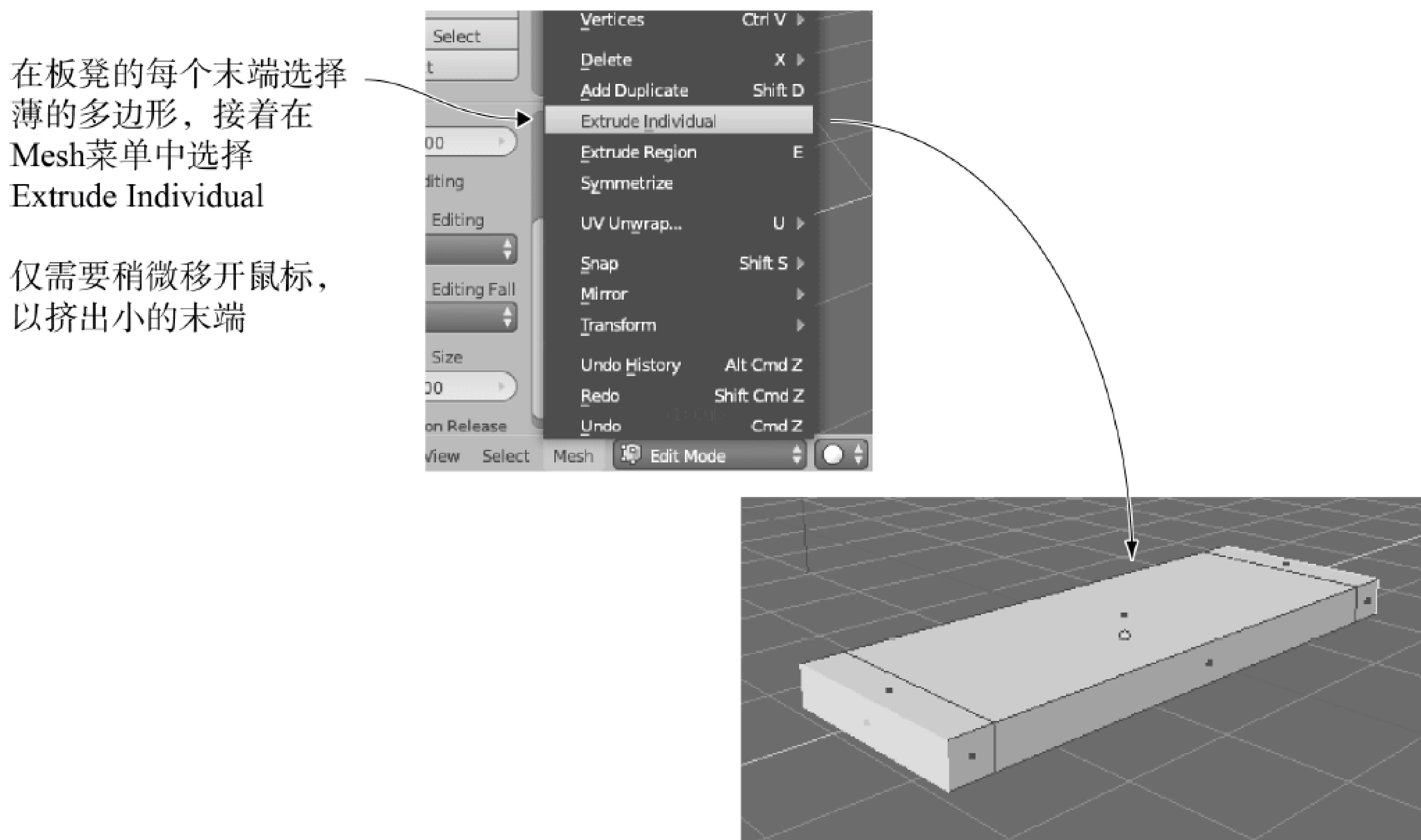


图 C-6　在 Mesh 菜单中使用 Extrude Individual 以挤出额外部分

定义 Extrude 会在图形上选中的面的相交部分挤出新的几何体。两个不同的挤出命令定义了当选择多个元素时应该做什么：Extrude Individual 把每个元素作为一个独立的部分挤出，而 Extrude Region 把整个选中元素作为一个独立的块挤出。

现在查看木板底部，并在两端选择两个薄的面。再次使用 Extrude Individual 命令，拉下板凳的桌脚(如图 C-7 所示)。

形状已经完成！但在将模型导出到 Unity 之前，需要考虑模型的贴图。

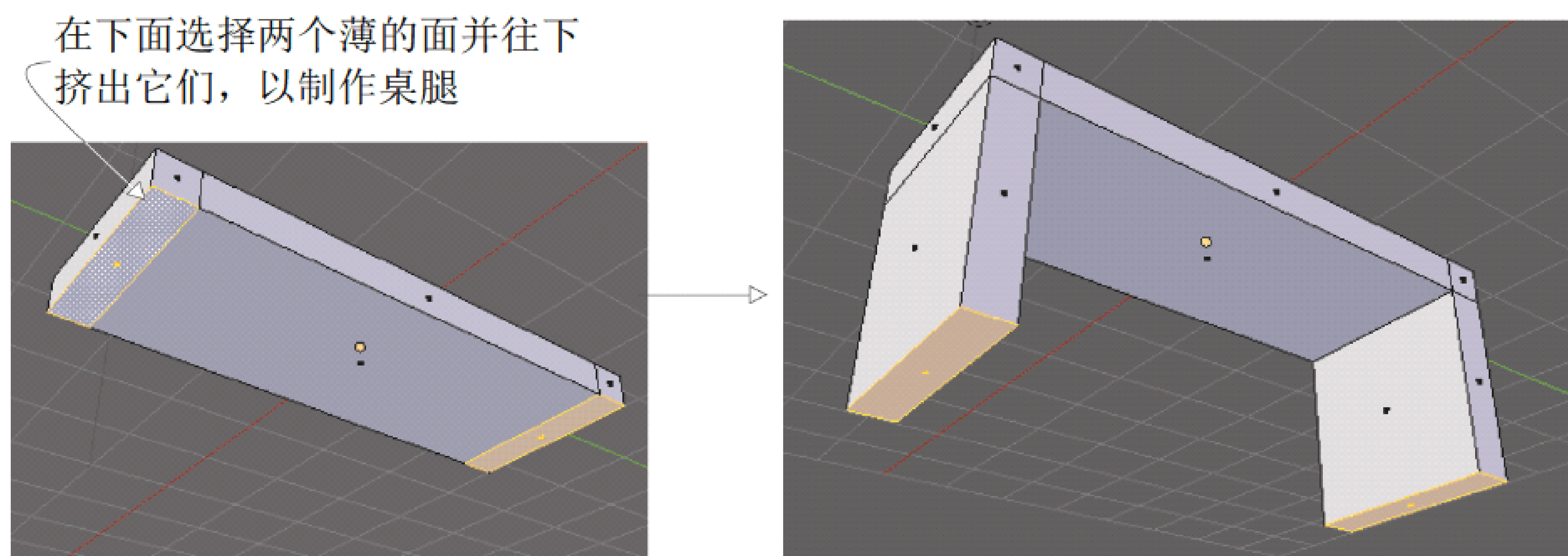


图 C-7　选择板凳下面的薄面并拉下桌腿

C.2　模型的贴图映射

3D 模型可以在其表面显示 2D 图像(称为贴图)。对于像墙一样的大平面来说，2D 图像与 3D 表面的关系是很直接的：简单地将图像拉伸到平面上。但是，像长凳两边这样形状奇特的表面，该怎么办呢？了解贴图坐标这个概念是重要的。

贴图坐标定义了贴图的各部分是如何与网格的各部分关联的。这些坐标将网格元素分配到贴图的区域。想象一下包装纸(如图 C-8 所示)，3D 模型是要包起来的盒子，贴图是包装纸，贴图坐标表示包装纸上的哪个位置将贴到盒子上的哪一面。贴图坐标定义了 2D 图像上的点和图形，这些图形关联到网格上的多边形，贴图坐标指定了图像的哪部分应该出现在网格的哪部分上。

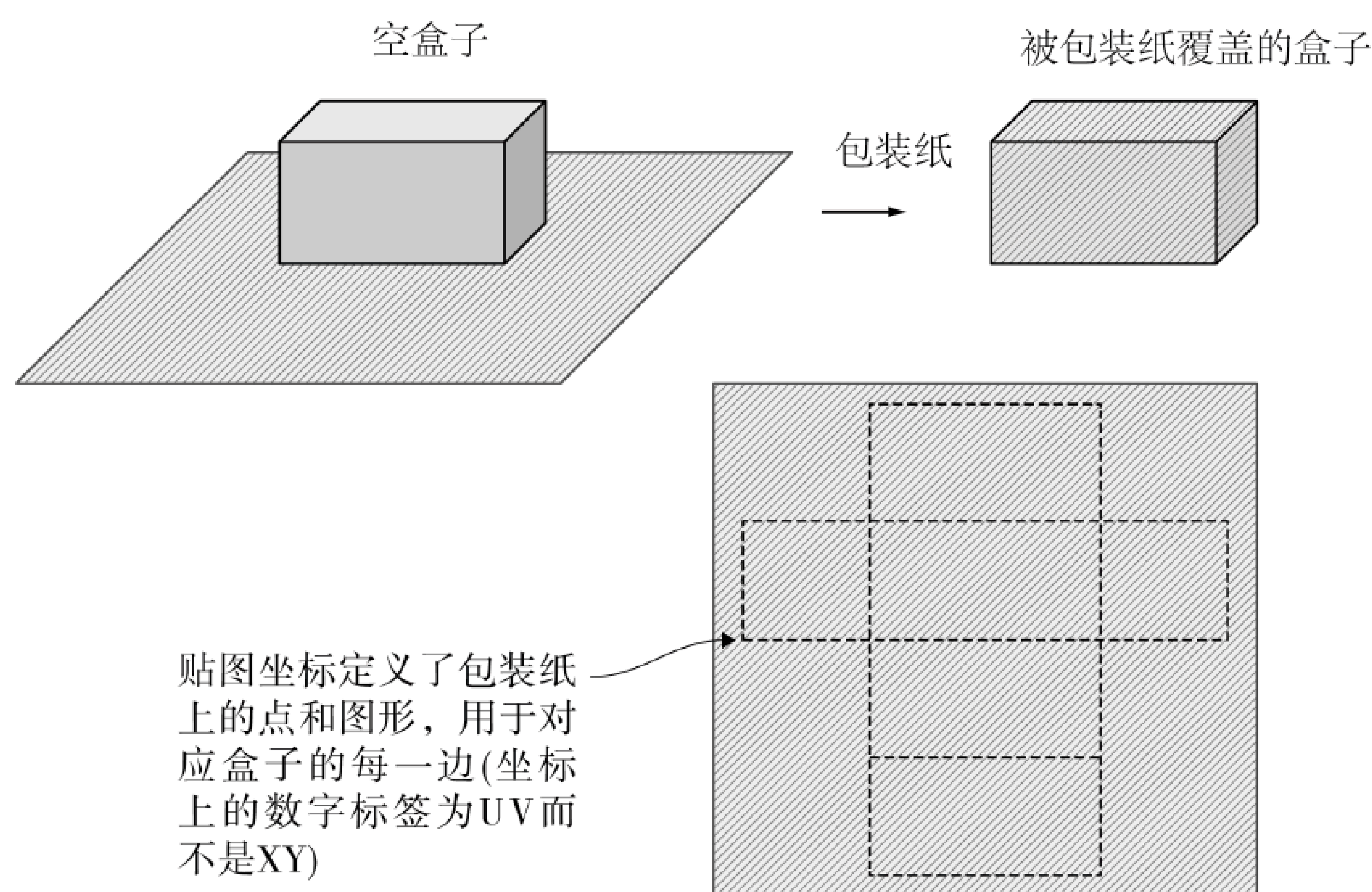


图 C-8　包装纸为贴图坐标的工作原理提供了恰当的类比

提示 贴图坐标也称为 UV 坐标。这个名称源于贴图坐标使用字母 U，V 定义的事实，类似 3D 模型坐标的定义使用 X，Y，Z。

将一个对象的某部分关联到另一个对象的某部分上的技术术语称为映射(mapping)——因此术语贴图映射表示创建贴图坐标的过程。在包装纸的类比中，这个过程的另一个名称是展开。还有其他一些混合的技术术语，比如 UV 展开。贴图映射有很多本质上同义的术语，因此不要搞混它们。

传统上，贴图映射的过程非常复杂，但幸运的是，Blender 提供了一些工具，使这个过程相当简单。首先在模型上定义接缝。如果进一步考虑如何包装一个盒子(或者考虑另一个方向，展开一个盒子)，就会发现，并不是三维形状的每个部分都能在二维空间中保持无缝。沿着三维形状的边缘展开时，一定会接缝。Blender 允许选择边缘，并将它们声明为接缝。

切换到 Edge Selection 模式(如图 C-4 所示的按钮)，选择板凳底部的外边缘。现在选择 Mesh | Edges | Mark Seam(如图 C-9 所示)。这让 Blender 分离板凳的底部，以进行贴图映射。为板凳的边缘做相同的操作，但不要完全分离边缘，只需要分离沿着板凳脚向上的边缘。通过这种方式，当展开板凳时，边缘将会保持到板凳的连接。

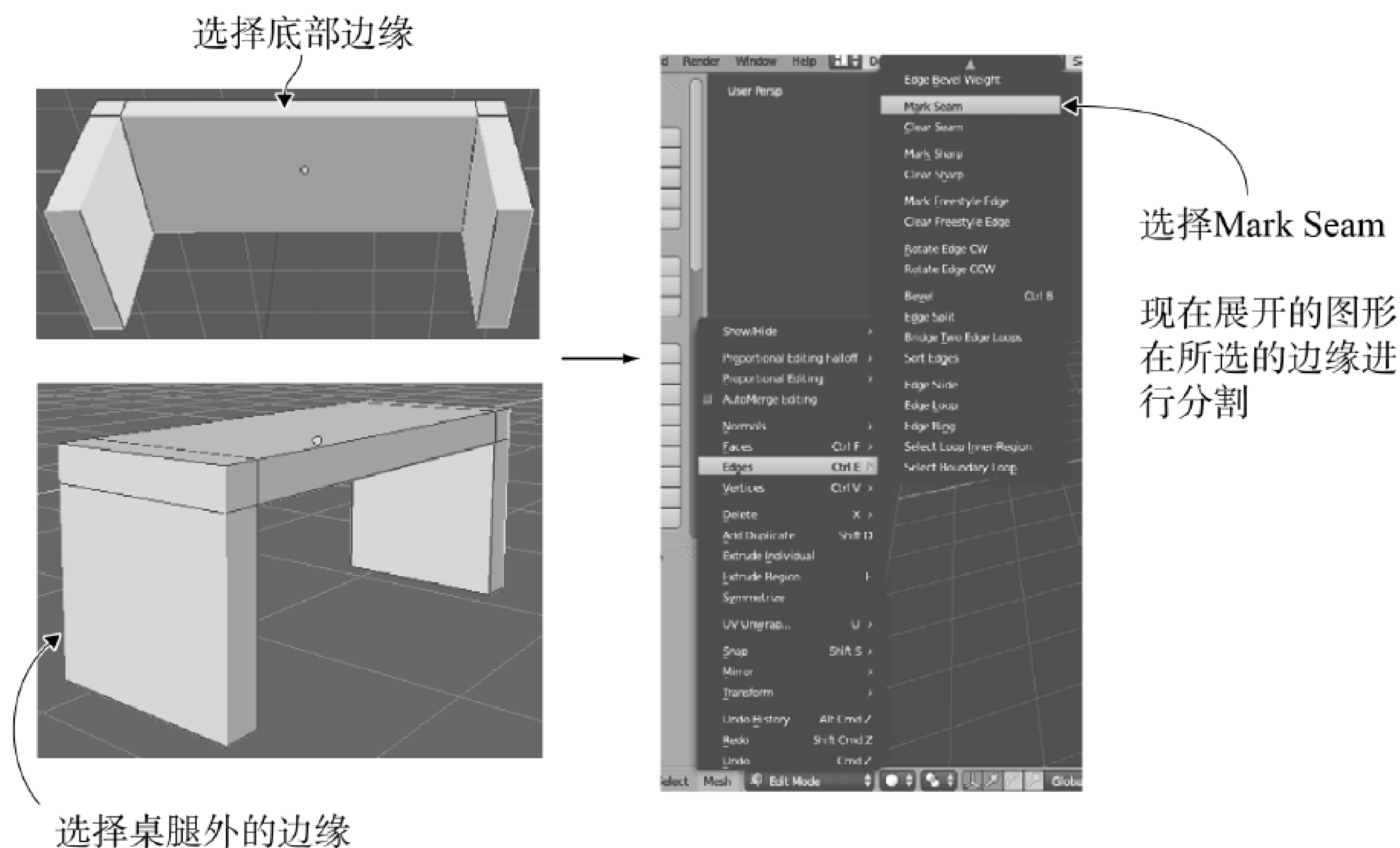


图 C-9　沿着板凳底部和板凳脚的边缘分离

一旦所有的接缝都已标记，就运行 Texture Unwrap 命令。首先选择整个网格(不要忘记对象的另一边)。接下来，选择 Mesh | UV Unwrap | Unwrap，创建贴图坐标。但在这个视图中看不到贴图坐标，Blender 默认显示场景的 3D 视图。为了看到贴图坐标，必须使用位于工具栏最左边的 Viewports 菜单(不是视图，而是一个小图标，如图 C-10 所示)，从 3D View 切换为 UV Editor。

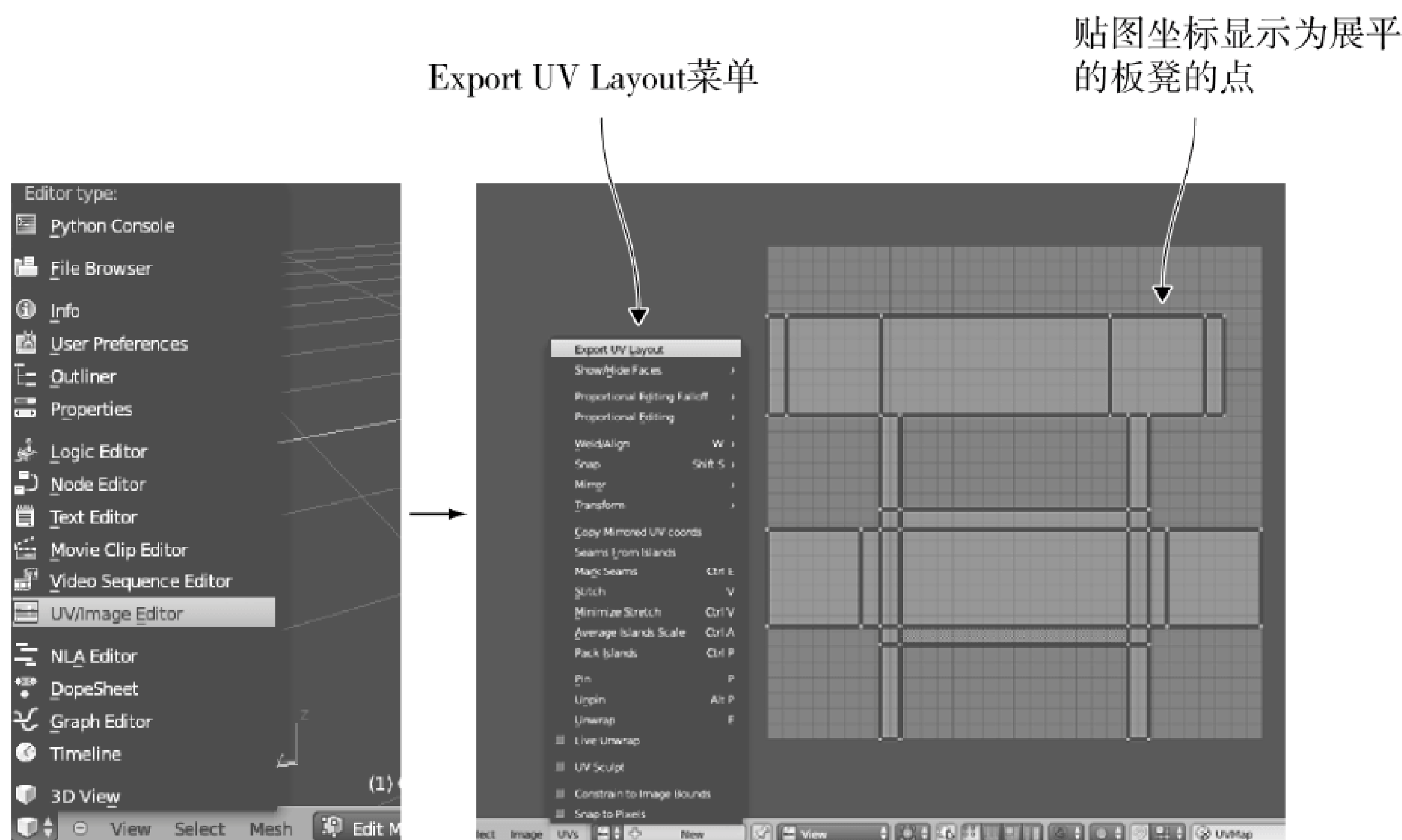


图 C-10　从 3D View 切换为 UV Editor，以显示贴图坐标

现在可以看到贴图坐标。可以看到板凳根据所标记的接缝展开为平面的多边形。为了绘制贴图，需要在图像编辑程序中查看这些 UV 坐标。再次参考图 C-10，在 UV 菜单下选择 Export UV Layout，将图像保存为 bench.png(这个名称将在以后导入 Unity 时使用)。

在图像编辑器中打开这个图像，为贴图的不同部分绘制颜色。为不同的 UV 绘制不同颜色，这样各个面上就会出现不同的颜色。例如，图 C-11 显示了深蓝色是在 UV Layout 顶部被展开的板凳的底部。板凳的侧面显示红色。现在可以将图像返回 Blender，给模型贴图，选择 Image | Open Image。

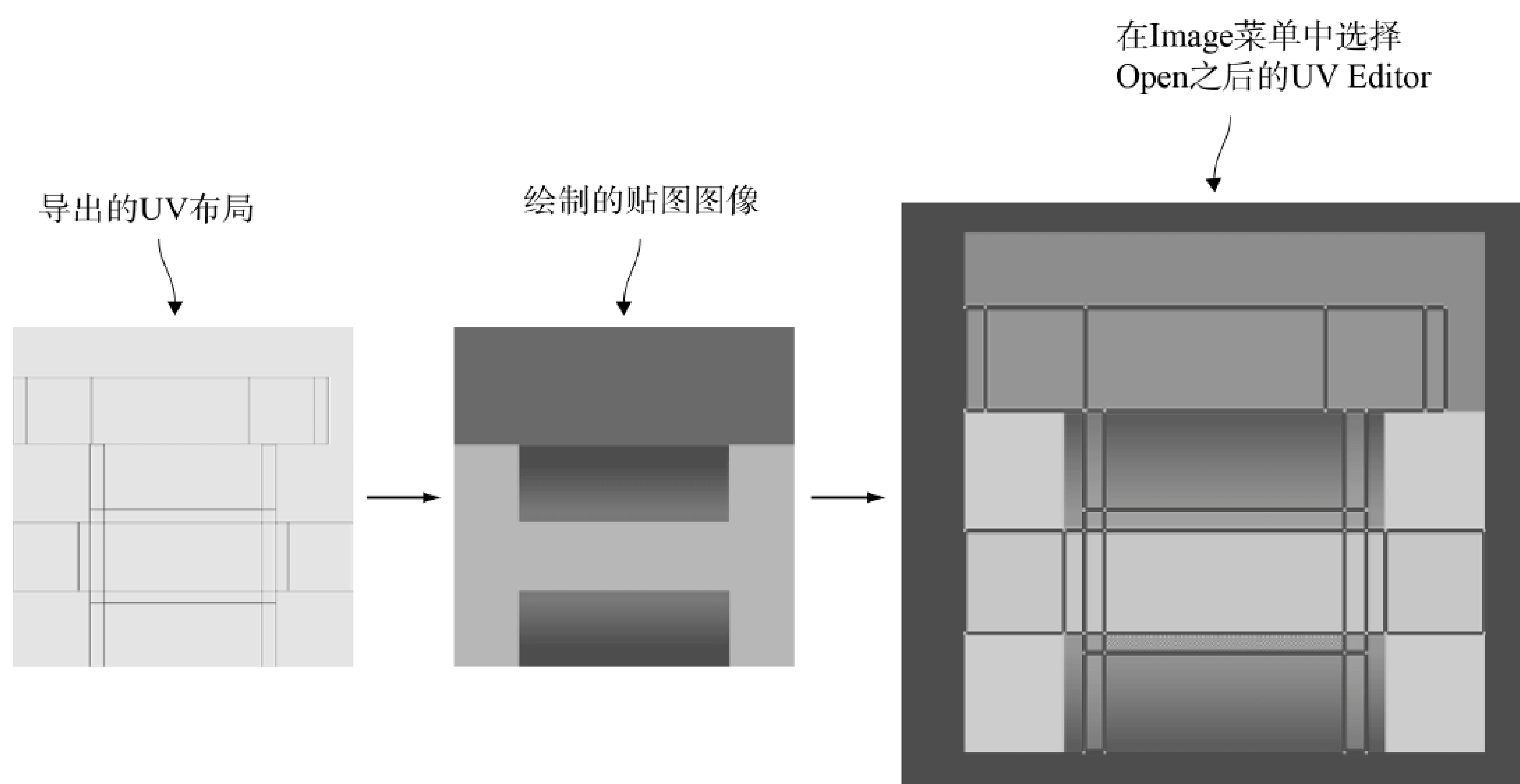
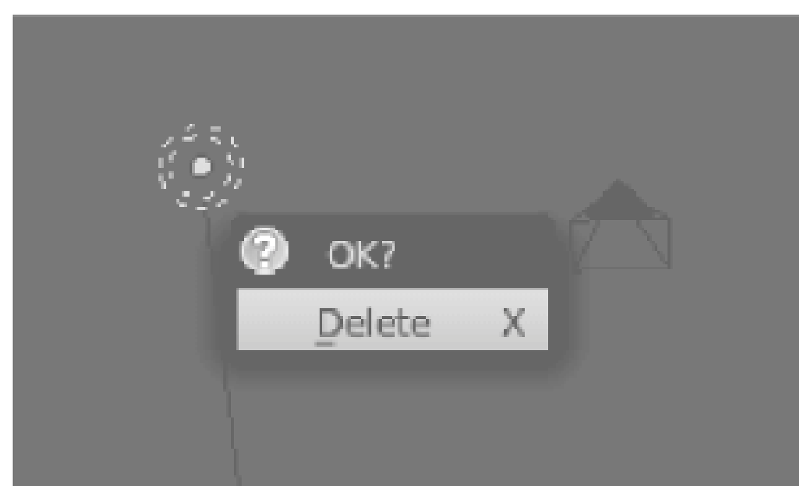


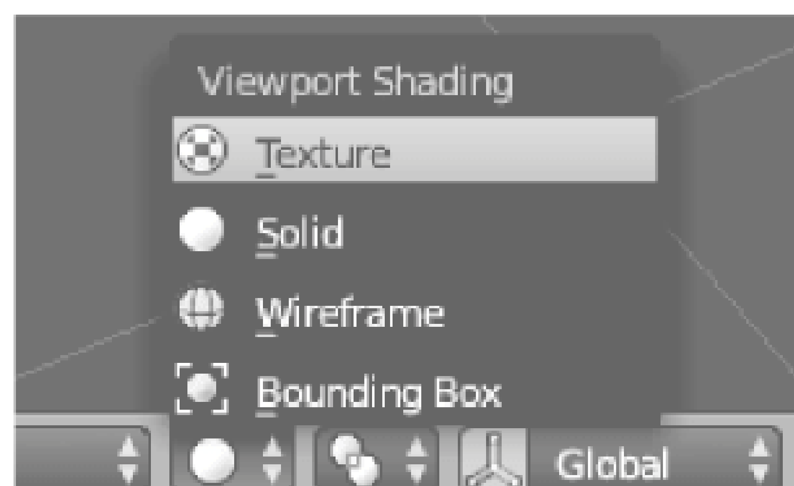
图 C-11　在导出的 UV 上绘制颜色，接着将贴图返回 Blender 中

此时，可以返回 3D 视图(使用切换到 UV Editor 的相同菜单)。模型上依然看不到贴图，但这只需要几步操作。需要删除默认的灯光，并打开视口中的贴图(如图 C-12 所示)。

1. 返回为Object模式并删除灯光(和摄像机)，单击X删除



2. 接着将Viewport Shading切换为Texture



3. 完成

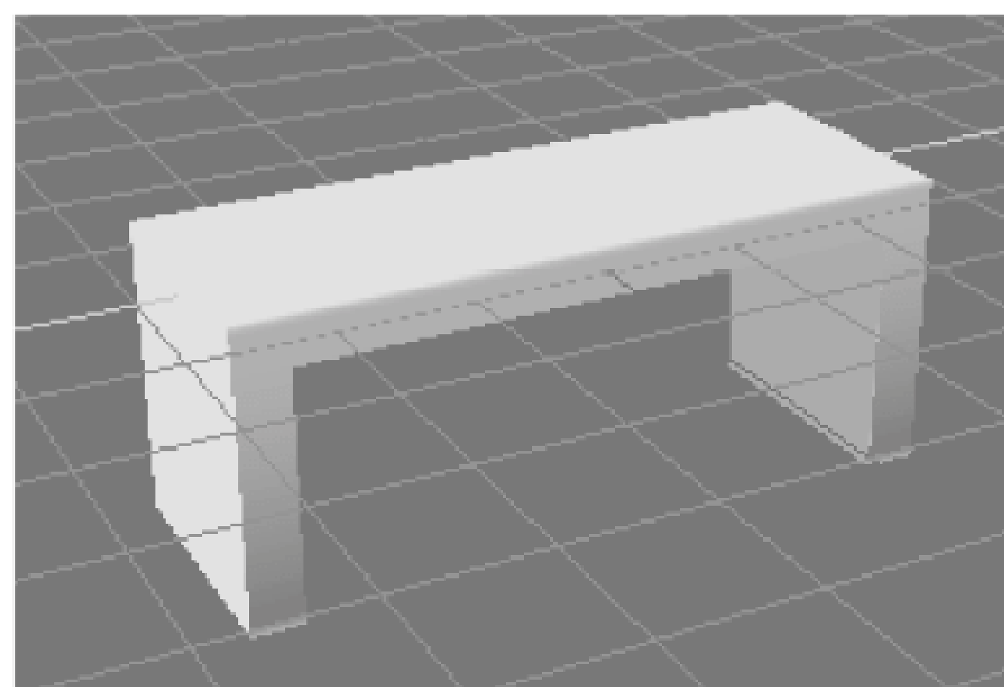


图 C-12　删除默认灯光，查看模型上的贴图

为了删除灯光，首先切换回 Object 模式以选中它(使用切换为 Edit 模式的菜单)。按下 X 键删除选中的对象，这也会删除摄像机。最后，将 Viewport Shading 菜单切换

为 Texture。现在就可以看到完成的板凳，也应用了贴图！

现在保存模型。Blender 将使用.blend 扩展名保存文件，为 Blender 使用本地文件格式。使用本地文件格式可以正确保存 Blender 的所有特性，但之后必须导出为不同的文件格式，以导入到 Unity 中。注意，贴图图像实际上没有保存在模型文件中，只是保存了对图像的引用，所以依然需要被引用的图像文件。

附录 D 在线学习资源



本书旨在完整介绍 Unity 中的游戏开发，但还有许多要学习的内容。下面有很多好的在线资源可以在完成本书之后使用。

D.1 其他指南

很多网站提供了 Unity 中各种话题的指导信息。其中一些甚至由 Unity 背后的公司官方提供。

Unity 手册

这是 Unity 提供的综合用户手册。它不仅有助于查找信息，还提供了 Unity 完整功能的主题列表：http://docs.unity3d.com/Documentation/Manual/index.html。

脚本参考

Unity 编程人员比其他人更应该读完这个脚本参考。用户手册覆盖了引擎的功能和编辑器的用法，但脚本参考是 Unity 完整 API 的全面参考。每个 Unity 命令都列在其中：http://docs.unity3d.com/Documentation/ScriptReference/ index.html。

Unity 学习教程

Unity 的官方网站包括几个综合教程，可以在学习部分找到。最重要的是，这些教程都是视频。根据读者自己的喜好，这可能是好事，也可能是坏事。如果喜欢看视频教程，就可以访问 https://unity3d.com/learn/tutorials。

Catlike Coding

Catlike Coding 提供了许多有用、有趣的话题，而不是让学习者通过一个完整的游戏来学习。这些主题甚至不一定是关于游戏开发的，却是在 Unity 中掌握编程技能的好方法。这些教程可以在 catlikecoding.com/unity/tutorials/上找到。

StackExchange 的游戏开发

StackExchange 是另一个很好的信息站点，其格式与前面列出的不同。StackExchange 没有提供一系列自包含的教程，而是提供了一个鼓励搜索的文本 QA。StackExchange 包含大量的主题，https://gamedev.stackexchange.com/是该网站专注于游戏开发的区域。以后在那里寻找 Unity 信息的频率几乎和使用脚本参考的频率一样高。

Maya LT Guide

如附录 B 中所述，外部美术应用是创建优秀可视化游戏的重要部分。有很多资源讲授 Maya、3ds Max、Blender 或其他外部 3D 美术应用。附录 C 是关于 Blender 的一个教程。下面是使用 MayaLT 的一个在线向导(这是一个稍微便宜且面向游戏开发的 Maya 版本)：http://steamcommunity.com/sharedfiles/filedetails/?id=242847724。

D.2 代码库

尽管前面列出的资源提供了 Unity 的教程和/或学习信息，本节列出的站点提供了可用于项目的代码。对于新手而言，库和插件是另一种类型的有用资源，它们不仅可以直接使用，还可以作为学习资源(通过阅读它们的代码)。

Unify Community Wiki

这个 Unify wiki(称为 Unify)是一个中心数据库，包括很多开发者贡献的代码，该库中的脚本覆盖的功能很广。本书有时使用该库中的脚本(例如，消息系统)。还有很多有用的脚本也可以在这个库中找到：http://wiki.unity3d.com/index.php/Scripts。

DOTween, LeanTween, iTween

如第 3 章所述，常用于游戏的一种运动效果称为缓动(tween)。在这种运动类型中，一个代码命令可以设置对象在一定的时间内移动到目标。缓动功能可以通过一些库(如 dotween.demigiant.com/、https://github.com/dentedpixel/LeanTween 和 www.itween.pixelplacement.com/)添加到 Unity 中。

Post-processing Stack

后期处理堆栈是一种简单的方法，可以为游戏添加一些视觉效果，比如景深和动态模糊。这些效果以前都是在 Unity 中单独提供的，但是现在它们整合到一个 über 效果中。这个组件可以在 Asset Store 中使用，也可以从 https://github.com/Unity-Technologies/PostProcessing 下载。

Prime[31]

Unity 为像 iOS 和 Android 这样的移动平台提供了部署，但是平台特有的功能仅限于核心功能。可以通过插件添加许多特定的特性，prime[31] (https://prime31.com/)有许多这样的插件。

Agasper Android Notifications

prime[31]的插件涵盖了各种不同的特性，而 Agasper Android Notifications (https://github.com/Agasper/unity-android-notifications)只关注 Android 上的本地通知。由于 Unity 内置了 iOS 通知，因此只需要插件的 Android 通知，在这种情况下，这是一个精简的选项。

Google 的 Play Games Services

在 iOS 系统上，Unity 内置了 GameCenter，这样游戏就可以拥有平台自带的排行榜和成就。Android 上的类似系统称为 Google Play Games。虽然它没有内置到 Unity 中，但是谷歌在 https://github.com/playgameservices/play-games-plugin-for-unity 上有一个插件。

FMOD Studio

Unity 内置的音频功能可以很好地播放录音，但是对于高级的声音设计工作来说，存在一定的局限。FMOD Studio 是一个高级的声音设计工具，有一个 Unity 插件。网址是 www.fmod.com/studio。

后　　序

至此，你已经学会了使用 Unity 构建完整游戏所需的一切知识——这里的“一切”是从编程的角度而言。顶级游戏也需要完美的画面和声音，所以游戏开发者的成功不仅包含技术技能。学习 Unity 并不是最终的目标，最终的目标是创建成功的游戏，而 Unity 仅仅是达成该目标的工具(却是一个很好的工具)。

除了实现整个游戏的技术技能外，还需要另一个无形的属性：决心。其含义是对挑战性的项目要坚持不懈并抱有自信，坚持到底，这有时指的是“完成能力”。只有一种方式可以提升完成能力，那就是完成很多项目。这似乎是互相矛盾的(为了获取完成项目的能力，首先需要完成很多项目)，但需要意识到的关键点是，小项目比大项目更容易完成。

因此，前进的路径是先构建很多小项目——因为小项目更容易完成——接着逐渐开始更大的项目。很多游戏开发新手都会犯的错误是，所构建的项目太大了。这有两个主要原因：他们想复制自己喜欢的(大)游戏，而每个人却低估了制作游戏所需要的工作量。项目刚开始看起来很好，但很快就面对太多的挑战，而最后开发者非常沮丧，放弃开发。

游戏开发新手应该从小项目开始。但项目太小，会令人觉得项目不重要，本书中的项目就是“小且几乎不重要”的类型，应该从它开始。如果完成了本书所有的项目，就掌握了很多额外的知识。接下来尝试大一些的项目，但一定要谨慎，不能跳跃性太大。这样就可以提升技能和自信，在每次开发项目时都可以更有雄心。

只要询问如何开始开发游戏，都会听到这个建议。例如 Unity 会请求网页系列 *Extra Credits*(一个关于游戏开发的经典系列)，制作一些关于开始游戏开发的视频，通过下面的网址可以找到这些视频：

http://unity3d.com/learn/tutorials/modules/beginner/your-first-game/

游戏设计

整个 *Extra Credits* 系列不仅包含由 Unity 发起的少数视频，还涵盖了很多领域，但大多数关注游戏的设计。

定义　游戏设计通过设定游戏目标、规则和挑战来定义游戏的过程。不应将游戏设计与可视化设计相混淆，可视化设计指的是设计外观，而不是功能。这是一个常见的错误，因为普通人对“设计”最熟悉的理解是“图形设计”。

定义 游戏设计最核心的一个部分是制作游戏机制，这是游戏中独立的行为(或者是行为系统)。游戏机制通常由它的规则创建，而游戏中的挑战通常来自于对特定情况应用机制。例如，在游戏中移动是一种机制，而迷宫是基于该机制的一种挑战。

游戏设计对于游戏开发新手会很棘手。一方面，最成功的(最想创建的)游戏是使用有趣、新颖的游戏机制创建的。另一方面，过多担心第一个游戏的设计会令人无法关注游戏开发的其他方面，例如，学习如何编写游戏。最好通过模仿已有游戏的设计开始游戏设计(记住，这是指开始阶段，复制已有的游戏对于初始练习十分可行，但最终读者将有足够的技能和经验来进一步扩展它)。

也就是说，任何成功的游戏开发者应该对游戏设计有好奇心。可以通过很多方式来学习游戏设计——前面介绍了 *Extra Credits* 视频，还有一些其他的网站，如下所示：

- www.gamesutra.com——提供游戏的工作机会，游戏更新，关于游戏的好消息/坏消息，制作游戏的艺术和商业信息等。
- www.lostgarden.com——提供游戏设计理论、美术和设计业务等深思熟虑、值得一读的文章(引自其主页)。
- http://sloperama.com——单击 School-a-rama 获取游戏商业建议页面。

有很多关于游戏设计的优秀图书，如下所示：

- *Game Design Workshop*，Third Edition，Tracy Fullerton 撰著(A K Peters / CRC Press，2014)。
- *A Theory of Fun for Game Design*，Second Edition，Raph Koster 撰著(O'Reilly Media, 2013)。
- *The Art of Game Design*，Second Edition，Jesse Schell 撰著(A K Peters/CRC Press, 2014)。

销售游戏

Extra Credits 视频中的第四个视频是关于销售游戏的内容。有时候游戏开发者没有考虑销售。他们只是考虑构建游戏，而没有考虑销售游戏，但这种态度可能会导致游戏失败。世界上编写得最好的游戏如果没有人知道，也不算成功！

单词“销售(marketing)”通常也考虑广告，而如果有预算，那么为游戏做广告肯定是一种将它推入市场的方式。但也可以通过很多成本较低，甚至免费的方式来推广游戏。具体的方式会随着时间而改变，但该视频中提到的策略包括在 Twitter 上发表(或者在常见社交媒体上发表，而不仅仅是在 Twitter 上发表)游戏相关的内容，创建一个跟踪视频，在 YouTube 上和评论者、博客使用者共享。一定要坚持并尝试！

现在就开始创建一些出色的游戏。Unity 是一个得力工具，你已经学会了如何使用它。祝你在游戏开发旅程中好运！